Hands-On Python
A Tutorial Introduction for Beginners
Python 3.1 Version

Dr. Andrew N. Harrington
Computer Science Department, Loyola University Chicago

Contents

CHAPTER 1

Beginning With Python

1.1. Context

You have probablry used computers to do all sorts of useful and interesting things. In each application, the computer responds in different ways to your input, from the keyboard, mouse or a file. Still the underlying operations are determined by the design of the program you are given. In this set of tutorials you will learn to write your own computer programs, so you can give the computer instructions to react in the way *you* want.

1.1.1. Low-Level and High-Level Computer Operations. First let us place Python programming in the context of the computer hardware. At the most fundamental level in the computer there are instructions built into the hardware. These are very simple instructions, peculiar to the hardware of your particular type of computer. The instructions are designed to be simple for the hardware to execute, not for humans to follow. The earliest programming was done with such instructions. If was difficult and error-prone. A major advance was the development of higher-level languages and translators for them. Higher-level languages allow computer programmers to write instructions in a format that is easier for humans to understand. For example

```
z = x+y
```

is an instruction in many high-level languages that means something like:

(1) Access the value stored at a location labeled x
(2) Calculate the sum of this value and the value stored at a location labeled y
(3) Store the result in a location labeled z.

No computer understands the high-level instruction directly; it is not in machine language. A special program must first translate instructions like this one into machine language. This one high-level instruction might be translated into a sequence of three machine language instructions corresponding to the three step description above:

```
0000010010000001
0000000010000010
0000010110000011
```

Obviously high-level languages were a great advance in clarity!

If you follow a broad introduction to computing, you will learn more about the layers that connect low-level digital computer circuits to high-level languages.

1.1.2. Why Python. There are many high-level languages. The language you will be learning is Python. Python is one of the easiest languages to learn and use, while at the same time being very powerful: It is used by many of the most highly productive professional programmers. A few of the places that use Python extensively are Google, the New York Stock Exchange, Industrial Light and Magic, ... Also Python is a free language! If you have your own computer, you can download it from the Internet....

1.1.3. Obtaining Python for Your Computer. If you are not sure whether your computer already has Python, continue to Section 1.2.2, and give it a try. If it works, you are all set.

If you do need a copy of Python, go to the Downloads page linked to `http://www.python.org`. Be careful to choose the version for your operating system and hardware. Chosse a stable version, 3.1 or later. Do not choose a version 2.X, which is incompatible. (Version 2.6 is described in an older version of this tutorial.)

Windows	You just need to execute the installer, and interact enough to agree to all the default choices. Python works in Windows as well as on Apples and in the free operating system Linux.
OS X	Double-click on the installer. Find and run the MacPython.mpkg that is inside. Follow the defaults for installation.
Linux	Python is generally installed, though Idle is not always installed. Look for something like 'idle-python' (the name in the Ubuntu distribution).

1.1.4. Philosophy and Implementation of the Hands-On Python Tutorials. Although Python is a high-level language, it is *not* English or some other natural human language. The Python translator does not understand "add the numbers two and three". Python is a formal language with its own specific rules and formats, which these tutorials will introduce gradually, at a pace intended for a beginner. These tutorials are also appropriate for beginners because they gradually introduce fundamental logical programming skills. Learning these skills will allow you to much more easily program in other languages besides Python. Some of the skills you will learn are

- breaking down problems into manageable parts
- building up creative solutions
- making sure the solutions are clear for humans
- making sure the solutions also work correctly on the computer.

Guiding Principals for the Hands-on Python Tutorials:

- The best way to learn is by active participation. Information is principally introduced in small quantities, where your active participation, experiencing Python, is assumed. In many place you will only be able to see what Python does by doing it yourself (in a hands-on fashion). The tutorial will often not show. Among the most common and important words in the tutorial are "Try this:"
- Other requests are for more creative responses. Sometimes there are Hints, which end up as hyperlinks in the web page version, and footnote references in the pdf version. Both formats should encourage you to think actively about your response first before looking up the hint.

  The tutorials also provide labeled exercises, for further practice, without immediate answers provided. The exercises are labeled at three levels
  ***:** Immediate reinforcement of basic ideas – preferably do on your first pass.
  ****:** Important and more substantial – be sure you can end up doing these.
  *****:** Most creative
- Information is introduced in an order that gives you what you need as soon as possible. The information is presented in context. Complexity and intricacy that is not immediately needed is delayed until later, when you are more experienced.
- In many places there are complications that *are* important in the beginning, because there is a *common* error caused by a slight misuse of the current topic. If such a common error is likely to make no sense and slow you down, more information is given to allow you to head off or easily react to such an error.

Although this approach is an effective way to introduce material, it is not so good for reference. Referencing is addressed in several ways:

- An extensive Table of Contents
- Easy jumping to chosen text in a browser like Firefox
- Cross references to sections that elaborate on an introductory section
- Concise chapter summaries, grouping logically related items, even if that does not match the order of introduction.

Some people learn better visually and verbally from the very beginning. Some parts of the tutorial will also have links to corresponding flash video segments. Many people will find reading faster and more effective, but the video segments may be particularly useful where a computer interface can be not only explained but actually demonstrated. The links to such segments will be labeled. They will need a broadband link or a CD (not yet generated).

In the Firefox browser, the incremental find is excellent, and particularly useful with the single web page version of the tutorials. (It only fails to search footnotes.) It is particularly easy to jump through different sections in a form like 1.2.4.

1.2. The Python Interpreter and Idle, Part I

1.2.1. Your Python Folder and Python Examples.

First you need to set up a location to store your work and the example programs from this tutorial. If you are on a Windows computer, follow just *one* of the three choices below to find an appropriate place to download the example archive examples.zip, and then follow the later instructions to unzip the archive.

Your Own Computer: If you are at your own computer, you can put the folder for your Python programs most anywhere you like. For Chapter 4, it will be important that none of the directories leading down to your Python folder contain any blanks in them. In particular in Windows, "My Documents" is a bad location. In Windows you can create a directory in C: drive, like C:\myPython. You should have installed Python to continue.

Your Flash Drive: If you do not have your own computer, or you want to have your materials easily travel back and forth between the lab and home, you will need a flash drive.

Plug your flash drive into the computer USB port.

On the computers in the Loyola lab DH 342, you can attach to the end of a cable that reaches close to the keyboard. In DH 339, there are USB ports on the monitor. Please Note: Flash drives are easy for me to forget and leave in the computer. I have lost a few this way. If you are as forgetful as I, you might consider a string from the flash drive to something you will not forget to take with you.

Open *My Computer* (on the desktop) to see where the flash drive is mounted, and open that drive.

Temporary: If you (temporarily) do not have a flash drive and you are at a Loyola lab computer: Open *My Computer* from the desktop, and then select drive *D:*. Create a folder on drive D: with your name or initials to make it easy for you to save and remove things. Change to that folder. You should place the examples archive here. You will need to save your work somehow before you log off of the computer. You may want to email individual files to yourself, or rezip the examples folder and send just the one archive file to yourself each time until you remember a flash drive!

In Windows, after you have chosen a location for the archive, examples.zip, download it by *right* clicking on `http://cs.luc.edu/anh/python/hands-on/3.0/examples.zip` and selecting "Save As" or the equivalent on your browser and then navigate to save the archive to the chosen location on your computer. Note the the examples, like this version of the tutorial, are for Python 3.1. There were major changes to Python in version 3.0, making it incompatible with earlier versions.

If you are using Python version 2.5 or 2.6, you should continue with the older version of the tutorial. Go to `http://cs.luc.edu/~anh/python/hands-on` and find the links to the proper version of the tutorial and examples.

Once you have the archive, open a file browser window for that directory, right click on examples.zip, select Extract All. This will create the folder `examples`. End up with a file browser window showing the contents of the examples folder. This will be your *Python folder* in later discussion.

Caution 1: On Windows, files in a zip archive can be viewed while they are still in the zip archive. Modifying and adding files is not so transparent. Be sure that you unzip the archive and work from the regular directory that holds the resulting unzipped files.

Caution 2: Make sure that all the directories leading down to your Python examples directory do not include any *spaces* in them. This will be important in Chapter 4 for the local webserver. In particular, that means you should *not* place your folder under "My Documents". A directory like C:\hands-on or C:\python would be fine.

You also have the option of downloading

- An archive containing the web version of the tutorial `http://cs.luc.edu/anh/python/hands-on/3.0/handsonHtml.zip` for local viewing, without the Internet. Download it and unzip as with the

examples. The local file to open in your browser in in handsonHtml folder you unzipped and the main web page file to open is called handson.html.

- The PDF version of the tutorial for printing `http://cs.luc.edu/anh/python/hands-on/3.0/handson.pdf`.

The disadvantage of a local copy is that the tutorial may be updated online after you get your download. The change log file `http://www.cs.luc.edu/~anh/python/hands-on/changelog.html` will show when the latest update was made and a summary of any major changes.

1.2.2. Running A Sample Program.

This section assumes Python, version at least 3.1, is already on your computer. Windows does not come with Python. (To load Python see Section 1.1.2) On a Mac or Linux computer enough of Python comes installed to be able to run the sample program.

If you are in a Windows lab with Python 3.1 installed, but not set up as the default version, see the footnote.[1]

Before getting to the individual details of Python, you will run a simple text-based sample program. Find `madlib.py` in your Python folder (Section 1.2.1).

Options for running the program:

- In Windows, you can display your folder contents, and double click on madlib.py to start the program.
- In Linux or on a Mac you can open a terminal window, change into your python directory, and enter the command

  ```
  python madlib.py
  ```

The latter approach only works in a Windows command window if your operating system execution path is set up to find Python.

In whatever manner you start the program, run it, responding to the prompts on the screen. Be sure to press the enter key at the end of each response requested from you.

Try the program a second time and make different responses.

1.2.3. A Sample Program, Explained. If you want to get right to the detailed explanations of writing your own Python, you can *skip to the next section* 1.2.4. If you would like an overview of a working program, even if all the explanations do not make total sense yet, read on.

Here is the text of the madlib.py program, followed by line-by-line brief explanations. Do not worry if you not totally understand the explanations! Try to get the gist now and the details later. The numbers on the right are not part of the program file. They are added for reference in the comments below.

```
1  """
2  String Substitution for a Mad Lib
3  Adapted from code by Kirby Urner
4  """
5  
6  storyFormat = """
7  Once upon a time, deep in an ancient jungle,
8  there lived a {animal}.  This {animal}
9  liked to eat {food}, but the jungle had
10 very little {food} to offer.  One day, an
11 explorer found the {animal} and discovered
12 it liked {food}.  The explorer took the
13 {animal} back to {city}, where it could
14 eat as much {food} as it wanted.  However,
15 the {animal} became homesick, so the
```

[1]If an earlier version of Python is the default in your lab (for instance Python 2,6), you can open the examples folder and double-click on the program `default31.cmd`. This will make Python 3.1 be the default version until you log out or reboot. This is only actually important when you run a Python program directly from a Windows folder. You will shortly see how to start a program from inside the Idle interactive environment, and as long as you run all your programs inside that environment, the system default version is not important.

```
16 explorer brought it back to the jungle,
17 leaving a large supply of {food}.
18
19 The End
20 """
21
22 def tellStory():
23     userPicks = dict()
24     addPick('animal', userPicks)
25     addPick('food', userPicks)
26     addPick('city', userPicks)
27     story = storyFormat.format(**userPicks)
28     print(story)
29
30 def addPick(cue, dictionary):
31     '''Prompt for a user response using the cue string,
32     and place the cue-response pair in the dictionary.
33     '''
34     prompt = 'Enter an example for ' + cue + ': '
35     response = input(prompt)
36     dictionary[cue] = response
37
38 tellStory()
39 input("Press Enter to end the program.")
```

Line By Line Explanation

```
1 """
2 String Substitution for a Mad Lib
3 Adapted from code by Kirby Urner
4 """
```

1-4 There is multi-line text enclosed in triple quotes. Quoted text is called a *string*. A string at the very beginning of a file like this is *documentation* for the file.

5,21,29,37 Blank lines are included for human readability to separate logical parts. The computer ignores the blank lines.

```
6 storyFormat = """
7 Once upon a time, deep in an ancient jungle,
8 there lived a {animal}.  This {animal}
9 liked to eat {food}, but the jungle had
10 very little {food} to offer.  One day, an
11 explorer found the {animal} and discovered
12 it liked {food}.  The explorer took the
13 {animal} back to {city}, where it could
14 eat as much {food} as it wanted.  However,
15 the {animal} became homesick, so the
16 explorer brought it back to the jungle,
17 leaving a large supply of {food}.
18
19 The End
20 """
```

6 The equal sign tells the computer that this is an *assignment statement*. The computer will now associate the value of the expression between the triple quotes, a multi-line *string*, with the name on the left, `storyFormat`.

7-20 These lines contain the body of the string and the ending triple quotes. This `storyFormat` string contains some special symbols making it a *format string*, unlike the string in lines 1-4. The

`storyFormat` string will be used later to provide a format into which substitutions are made. The parts of the string enclosed in braces are places a substitute string will be inserted later. The substituted string will come from a custom *dictionary* that will contain the user's definitions of these words. The words in the braces: {animal}, {food}, {city}, indicate that `"animal"`, `"food"`, and `"city"` are words in a dictionary. This custom dictionary will be created in the program and contain the user's definitions of these words. These user's definitions will be substituted later in the *format string* where each `{...}` is currently.

```
22 def tellStory():
23     userPicks = dict()
24     addPick('animal', userPicks)
25     addPick('food', userPicks)
26     addPick('city', userPicks)
27     story = storyFormat.format(**userPicks)
28     print(story)
```

22 *def* is short for *def*inition. This line is the heading of a *def*inition, which makes the name `tellStory` becomes *def*ined as a short way to refer to the sequence of statements that start indented on line 23, and continue through line 27.

23 The equal sign tells the computer that this is another assignment statement. The computer will now associate the name ***userPicks*** with a new empty dictionary created by the Python code `dict()`.

24-26 `addPick` is the name for a sequence of instructions defined on lines 29-31 for adding another definition to a dictionary, based on the user's input. The result of these three lines is to add definitions for each of the three words 'animal', 'food', and 'city' to the dictionary called userPicks.

27 Assign the name `story` to a string formed by substituting into `storyFormat` using definitions from the dictionary userPicks, to give the user's customized story.

28 This is where all the work becomes visible: Print the ***story*** string to the screen.

```
30 def addPick(cue, dictionary):
31     '''Prompt for a user response using the cue string,
32     and place the cue-response pair in the dictionary.
33     '''
34     prompt = 'Enter an example for ' + cue + ': '
35     response = input(prompt)
36     dictionary[cue] = response
```

30 This line is the heading of a definition, which gives the name ***addPick*** as a short way to refer to the sequence of statements indented on line 34-36. The name ***addPick*** is followed by two words in parenthesis, ***cue*** and ***dictionary***. These two words are associated with an actual cue word and dictionary given when this definition is invoked in lines 24-26.

31-33 A documentation comment for the `addPick` definition.

34 The plus sign here is used to concatenate parts of the string assigned to the name `prompt`. The current value of `cue` is placed into the string.

35 The right-hand-side of this equal sign causes an interaction with the user. The prompt string is printed to the computer screen, and the computer waits for the user to enter a line of text. That line of text then becomes a string inside the program. This string is assigned to the name `response`.

36 The left-hand-side of the equal sign is a reference to the definition of the cue word in the dictionary. The whole line ends up making the definition of the current cue word become the response typed by the user.

```
38 tellStory()
39 input("Press Enter to end the program.")
```

38 The definition of `tellStory` above does not make the computer do anything besides *remember* what the instruction `tellStory` means. It is only in this line, with the name, `tellStory`, followed by parentheses, that the whole sequence of remembered instructions are actually carried out.

39 This line is only here to accommodate running the program in Windows by double clicking on its file icon. Without this line, the story would be displayed and then the program would end, and Windows would make it immediately disappear from the screen! This line forces the program to continue being displayed until there is another response from the user, and meanwhile the user may look at the output from `tellStory`.

1.2.4. Starting Idle.

The program that translates Python instructions and then executes them is the *Python interpreter*.

This interpreter is embedded in a number of larger programs that make it particularly easy to develop Python programs. Such a programming environment is *Idle*, and it is a part of the standard distribution of Python.

Read the section that follows for your operating system:

Windows (Assuming you already have Python installed.) Display your Python folder. You should see icon for `Idle31Shortcut` (and maybe a similar icon with a number larger than 31 - ignore any other unless you know you are using that version of Python). Double click on the appropriate shortcut, and an Idle window should appear. After this the instructions are the same in any operating environment. It is *important* to start Idle through these in several circumstances. *It is best if it you make it a habit to use this shortcut.* For example the alternative of opening an existing Python program in Windows XP or Vista from Open With Idle in the context menu looks like it works at first but then fails *miserably* but *inexplicably* when you try to run a graphics program.

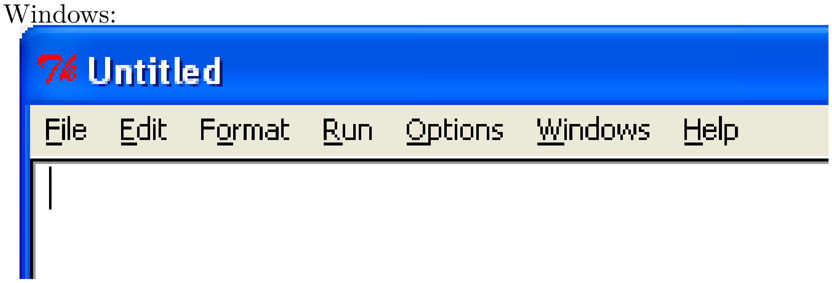
Mac OS X the new version of Python and Idle should be in a folder called MacPython 3.1, inside the Applications folder. It is best if you can open a terminal window, change into your Python folder from Section 1.2.1, and enter the command

```
idle
```

If the command is not recognized, you may need to include the full path to the idle program.

Linux The approach depends on the installation. In Ubuntu, you should fine idle in the Programming section of the Applications menu. As with OS X above, you are better starting idle from a terminal, with the current directory being your Python folder.

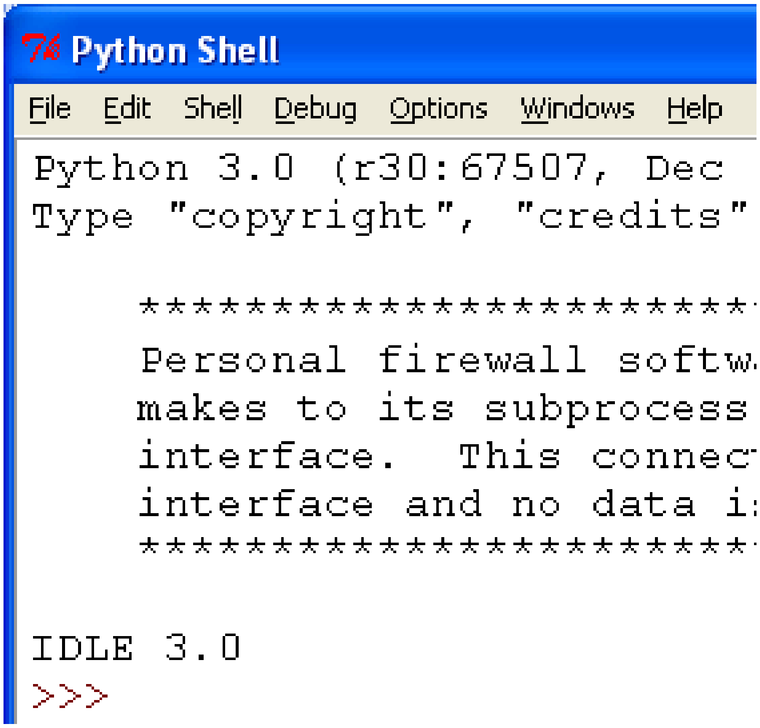
1.2.5. Windows in Idle. Idle has several parts you may choose to display, each with its own window. Depending on the configuration, Idle can start up showing either of two windows, an Edit Window or a Python Shell Window. You are likely to first see an Edit window, whose top left corner looks something like in Windows:



For more on the Edit Window, see Section1.9.

If you see this Edit Window with its Run menu on top, go to the Run menu and choose PYTHON SHELL to open a Python Shell Window for now. Then you may close the Edit Window.

Either initially, or after explicitly opening it, you should now see the Python Shell window, with a menu like the following, though the text may be slightly different:



Look at the Python Shell. ...

In the Shell the last line should look like

```
>>>
```

The `>>>` is the *prompt*, telling you Idle is waiting for you to type something. Continuing on the same line enter

```
6+3
```

Be sure to end with the Enter key. After the Shell responds, you should see something like

```
>>> 6+3
9
>>>
```

The shell evaluates the line you entered, and prints the result. You see Python does arithmetic. At the end you see a further prompt `>>>` where you can enter your next line.... The result line, showing `9`, that is produced by the computer, does not start with “>>>”.

1.3. Whirlwind Introduction To Types and Functions

Python directly recognizes a variety of types of data. Here are a few:

Numbers: 3, 6, -7, 1.25

Character strings: `'hello'`, `'The answer is: '`

Lists of objects of any type: `[1, 2, 3, 4]`, `['yes', 'no', 'maybe']`

A special datum meaning nothing: `None`

Python has large collection of built-in functions that operate on different kinds of data to produce all kinds of results. To make a function do its action, parentheses are required. These parentheses surround the parameter or parameters, as in a function in algebra class.

The general syntax to execute a function is

`functionName ( parameters )`

One function is called `type`, and it returns the type of any object. The Python Shell will evaluate functions. In the Shell the last line should look like

```
>>>
```

Continuing on the same line enter

```
type(7)
```

Always remember to end with the Enter key. After the Shell responds, you should see something like

```
>>> type(7)
<class 'int'>
>>>
```

In the result, `int` is short for integer. The work *class* is basically a synonym for type in Python. At the end you see a further prompt where you can enter your next line....

For the rest of this section, at the `>>>` prompt in the Python *Shell*, individually enter each line below that is set off in `typewriter` font. So next enter

```
type(1.25)
```

Note the name in the last result is `float`, not real or decimal, coming from the term "floating point", for reasons that will be explained later, in Section 1.14.1. Enter

```
type('hello')
```

In your last result you see another abbreviation: `str` rather than string. Enter

```
type([1, 2, 3])
```

Strings and lists are both sequences of parts (characters or elements). We can find the length of that sequence with another function with the abbreviated name `len`. Try both of the following, separately, in the *Shell*:

```
len([2, 4, 6])
len('abcd')
```

Some functions have no parameters, so nothing goes between the parentheses. For example, some types serve as no-parameter functions to create a simple value of their type. Try

```
list()
```

You see the way an empty list is displayed.

Functions may also take more than one parameter. Try

```
max(5, 11, 2)
```

Above, max is short for maximum.

Some of the names of types serve as conversion functions (where there is an obvious meaning for the conversion). Try each of the following, one at a time, in the *Shell*:

```
str(23)
int('125')
```

An often handy Shell feature: an earlier Shell line may to copied and edited by clicking anywhere in the previously displayed line and then pressing ENTER. For instance you should have entered several lines starting with `len`. click on any one, press ENTER, and edit the line for a different test.

1.4. Integer Arithmetic

1.4.1. Addition and Subtraction. We start with the integers and integer arithmetic, not because arithmetic is exciting, but because the symbolism should be mostly familiar. Of course arithmetic is important in many cases, but Python is probably more often used to manipulate text and other sorts of data, as in the sample program in Section 1.2.2.

Python understands numbers and standard arithmetic. *For the whole section on integer arithmetic*, where you see a set-off line in `typewriter` font, type individual lines at the >>> prompt in the Python *Shell*. Press Enter after each line to get Python to respond:

```
77
2 + 3
5 - 7
```

Python should evaluate and print back the value of each expression. Of course the first one does not require any calculation. It appears the shell just echoes back what you printed. Do note that the line with the value *produced* by the shell does not start with >>> and appears at the left margin. Hence you can distinguish what you type (after the ">>>" prompt) from what the computer responds.

The Python Shell is an interactive interpreter. As you can see, after you press Enter, it is evaluating the expression you typed in, and then printing the result automatically. This is a very handy environment to check out simple Python syntax and get instant feedback. For more elaborate programs that you want to save, we will switch to an Editor Window later.

1.4.2. Multiplication, Parentheses, and Precedence. Try in the *Shell*:

```
2 x 3
```

You should get your first *syntax* error. The 'x' should have become highlighted, indicating the location where the Python interpreter discovered that it cannot understand you: Python does not use x for multiplication as you may have done in grade school. The x can be confused with the use of x as a variable (more on that later). Instead the symbol for multiplication is an asterisk '*'. Enter each of the following. You may include spaces or not. The Python interpreter can figure out what you mean either way. Try in the *Shell*:

```
2*5
2 + 3 * 4
```

If you expected the last answer to be 20, think again: Python uses the normal *precedence* of arithmetic operations: Multiplications and divisions are done before addition and subtraction, unless there are parentheses. Try

```
(2+3)*4
2 * (4 - 1)
```

Now try the following in the *Shell*, exactly as written, followed by Enter, with *no* closing parenthesis:

```
5 * (2 + 3
```

Look carefully. There is no answer given at the left margin of the next line and no prompt >>> to start a *new* expression. If you are using Idle, the cursor has gone to the next line and has only *indented* slightly. Python is waiting for you to finish your expression. It is smart enough to know that opening parentheses are always followed by the same number of closing parentheses. The cursor is on a *continuation* line. Type just the matching close-parenthesis and Enter,

```
)
```

and you should finally see the expression evaluated. (In some versions of the Python interpreter, the interpreter puts '...' at the beginning of a continuation line, rather than just indenting.)

Negation also works. Try in the *Shell*:

```
-(2 + 3)
```

1.4.3. Division and Remainders. If you think about it, you learned several ways to do division. Eventually you learned how to do division resulting is a decimal. Try in the *Shell*:

```
5/2
14/4
```

As you saw in the previous section, numbers with decimal points in them are of type float in Python. They are discussed more in Section 1.14.1.

In the earliest grades you would say "14 divided by 4 is 3 with a remainder of 2". The problem here is that the answer is in two parts, the integer quotient 3 and the remainder 2, and neither of these results is the same as the decimal result. Python has separate operations to generate each part. Python uses the doubled division symbol `//` for the operation that produces just the integer quotient, and introduces the symbol `%` for the operation of finding the remainder. Try each in the *Shell*

```
14/4
14//4
14%4
```

Now predict and then try each of

```
23//5
23%5
20%5
6//8
6%8
```

Finding remainders will prove more useful than you might think in the future!

1.5. Strings, Part I

Enough with numbers for a while. Strings of characters are another important type in Python.

1.5.1. String Delimiters, Part I. A string in Python is a sequence of characters. For Python to recognize a sequence of characters, like `hello`, as a string, it must be enclosed in quotes to delimit the string.

For this whole section on strings, continue trying each set-off line of code in the *Shell*. Try

```
"hello"
```

Note that the interpreter gives back the string with *single* quotes. Python does not care what system you use. Try

```
'Hi!'
```

Having the choice of delimiters can be handy.

EXERCISE 1.5.1.1. * Figure out how to give Python the string containing the text: `I'm happy.` Try it. If you got an error, try it with another type of quotes, and figure out why that one works and not the first.

There are many variations on delimiting strings and embedding special symbols. We will consider more ways later in Section 1.8.

A string can have any number of characters in it, including 0. The empty string is `''` (two quote characters with nothing between them).

Strings are a new Python type. Try

```
type('dog')
type('7')
type(7)
```

The last two lines show how easily you can get confused! Strings can include any characters, *including* digits. Quotes turn even digits into strings. This will have consequences in the next section....

1.5.2. String Concatenation. Strings also have operation symbols. Try in the *Shell* (noting the *space* after `very`):

```
'very ' + 'hot'
```

The plus operation with strings means *concatenate* the strings. Python looks at the type of operands before deciding what operation is associated with the +.

Think of the relation of addition and multiplication of integers, and then guess the meaning of

```
3*'very ' + 'hot'
```

Were you right? The ability to repeat yourself easily can be handy.

EXERCISE 1.5.2.1. * Figure out a *compact* way to get Python to make the string, "YesYesYesYesYes", and try it. How about "MaybeMaybeMaybeYesYesYesYesYes" ? Hint: [2]

Predict the following and then test. Remember the last section on types:

```
7+2
'7'+'2'
```

Python checks the types and interprets the plus symbol based on the type. Try

```
'7'+2
```

With mixed string and int types, Python sees an ambiguous expression, and does not guess which you want – it just gives an error! [3]

---

[2]Hint for the second one: use two *'s and a +.

[3]Be careful if you are a Java programmer! This is unlike Java, where the 2 would be automatically converted to '2' so the concatenation would make sense.

1.6. Variables and Assignment

Each set-off line in this section should be tried in the Shell.

Try

```
width = 10
```

Nothing is displayed by the interpreter after this entry, so it is not clear anything happened. Something has happened. This is an *assignment statement*, with a *variable*, `width`, on the left. A variable is a name for a value. An assignment statement associates a variable name on the left of the equal sign with the value of an expression calculated from the right of the equal sign. Enter

```
width
```

Once a variable is assigned a value, the variable can be used in place of that value. The response to the expression `width` is the same as if its value had been entered.

The interpreter does not print a value after an assignment statement because the value of the expression on the right is not lost. It can be recovered if you like, by entering the variable name and we did above.

Try each of the following lines:

```
height = 12
area = width * height
area
```

The equal sign is an unfortunate choice of symbol for assignment, since Python's usage is not the mathematical usage of the equal sign. If the symbol ←had appeared on keyboards in the early 1990's, it would probably have been used for assignment instead of =, emphasizing the asymmetry of assignment. In mathematics an equation is an *assertion* that *both* sides of the equal sign *are already, in fact, equal*. A Python assignment statement *forces* the variable on the left hand side *to become* associated with the value of the expression on the right side. The difference from the mathematical usage can be illustrated. Try:

```
10 = width
```

so this is not equivalent in Python to `width = 10`. The *left hand* side must be a variable, to which the assignment is made. Try

```
width = width + 5
```

This is, of course, nonsensical as mathematics, but it makes perfectly good sense as an assignment, with the right-hand side calculated first. Can you figure out the value that is now associated with width? Check by entering

```
width
```

In the assignment statement, the expression on the right is evaluated *first*. At that point `width` was associated with its original value 10, so `width + 5` had the value of 10 + 5 which is 15. That value was then assigned to the variable on the left (`width` again) to give it a *new* value. We will modify the value of variables in a similar way routinely.

Assignment and variables work equally well with strings. Try:

```
first = 'Sue'
last = 'Wong'
name = first + ' ' + last
name
```

Try entering:

```
first = fred
```

Note the different form of the error message. The earlier errors in these tutorials were *syntax* errors: errors in translation of the instruction. In this last case the syntax was legal, so the interpreter went on to execute the instruction. Only *then* did it find the error described. There are no quotes around `fred`, so the interpreter assumed `fred` was an identifier, but the name fred was not defined at the time the line was executed.

It is easy to forget quotes where you need them and put them around a variable name that should not have them!

Try in the *Shell*:

```
fred = 'Frederick'
first = fred
first
```

Now `fred`, without the quotes, makes sense.

There are more subtleties to assignment and the idea of a variable being a "name for" a value, but we will worry about them later, in Section 2.4.6. They do not come up if our variables are just numbers and strings.

Autocompletion: A handy short cut. Python remembers all the variables you have defined at any moment. This is handy when editing. Without pressing Enter, type into the Shell just

f

Then *hold down* the Alt key and press the '/' key. This key combination is abbreviated Alt-/. You should see `f` autocompleted to be `first`. This is particularly useful if you have long identifiers! You can press Alt-/ several times if more than one identifier starts with the initial sequence of characters you typed. If you press Alt-/ again you should see `fred`. Backspace and edit so you have `fi`, and then and press Alt-/ again. You should not see fred this time, since it does not start with `fi`.

1.6.1. Literals and Identifiers. Expressions like `27` or `'hello'` are called *literals*, coming from the fact that they *literally* mean exactly what they say. They are distinguished from variables, who value is *not* directly determined by their name.

The sequence of characters used to form a variable name (and names for other Python entities later) is called an *identifier*. It identifies a Python variable or other entity.

There are some restrictions on the character sequence that make up an identifier:

- The characters must all be letters, digits, or underscores '_', and must start with a letter. In particular, punctuation and blanks are not allowed.
- There are some words that are *reserved* for special use in Python. You may not use these words as your own identifiers. They are easy to recognize in Idle, because they are automatically colored orange. For the curious, you may *read* the full list:

```
False    class     finally  is        return
None     continue  for      lambda    try
True     def       from     nonlocal  while
and      del       global   not       with
as       elif      if       or        yield
assert   else      import   pass
break    except    in       raise
```

There are also identifiers that are automatically defined in Python, and that you could redefine, but you probably should not unless you really know what you are doing! When you start the editor, we will see how Idle uses color to help you know what identifies are predefined.

Python is case sensitive: The identifiers `last`, `LAST`, and `LaSt` are all different. Be sure to be consistent. Using the Alt-/ auto-completion shortcut in Idle helps ensure you are consistent.

What is legal is distinct from what is conventional or good practice or recommended. Meaningful names for variables are important for the humans who are looking at programs, understanding them, and revising them. That sometimes means you would like to use a name that is more than one word long, like `price at opening`, but blanks are illegal! One poor option is just leaving out the blanks, like `priceatopening`. Then it may be hard to figure out where words split. Two practical options are

- underscore separated: putting underscores (which are legal) in place of the blanks, like `price_at_opening`.
- using *camelcase*: omitting spaces and using all lowercase, except capitalizing all words after the first, like `priceAtOpening`

Use the choice that fits your taste (or the taste or convention of the people you are working with).

1.7. Print Function, Part I

In interactive use of the Python interpreter, you can type an expression and immediately see the result of its evaluation. This is fine to test out syntax and maybe do simple calculator calculations. In a program

run from a file like the first sample program, Python does not display expressions this way. If you want your program to display something, you can give explicit instructions with the print function. Try in the *Shell*:

```
x = 3
y = 5
print('The sum of', x, 'plus', y, 'is', x+y)
```

The print function will prints as strings everything in a comma-separated sequence of expressions, and it will separate the results with single blanks by default. Note that you can mix types: anything that is not already a string is automatically converted to its string representation.

You can also use it with no parameters:

```
print()
```

to just advance to the next line.

1.8. Strings Part II

1.8.1. Triple Quoted String Literals. Strings delimited by one quote character are required to lie within a single Python line. It is sometimes convenient to have a multi-line string, which can be delimited with triple quotes: Try typing the following. You will get continuation lines until the closing triple quotes. Try in the *Shell*:

```
sillyTest = '''Say,
"I'm in!"
This is line 3'''
print(sillyTest)
```

The line structure is preserved in a multi-line string. As you can see, this also allows you to embed both single and double quote characters!

1.8.2. Escape Codes. Continuing in the *Shell* with `sillyTest`, enter just

```
sillyTest
```

The answer looks strange! It indicates an alternate way to encode the string internally in Python using *escape codes*. Escape codes are embedded inside string literals and start with a backslash character (\). They are used to embed characters that are either unprintable or have a special syntactic meaning to Python that you want to suppress. In this example you see the most common ones:

Escape code	Meaning
`\'`	'
`\n`	newline
`\\`	\

The newline character indicates further text will appear on a new line when *printed*. When you use a print function, you get the actual printed meaning of the escaped coded character.

Predict the result, and try in the *Shell*:

```
print('a\nb\n\nc')
```

Did you guess the right number of lines splitting in the right places?

1.9. The Idle Editor and Execution

1.9.1. Loading a Program in the Idle Editor, and Running It. It is time to put longer collections of instructions together. That is most easily done by creating a text file and running the Python interpreter on the file. Idle simplifies that process.

First you can put an existing file into an Idle Edit Window. Click on the Idle File menu and select Open. (Or as you see, you can use the shortcut Ctrl+O. That means holding down the Ctrl key, and pressing the letter O for Open.) You should get a file selection dialog. You should have the sample program madlib.py displayed in the list. Select it and open it. (If you do not see the program, then you either failed to download the example programs, Section 1.2.1, or you did not start Idle in the proper folder, Section 1.2.4.)

You will see the source code again. Now run this program from inside of Idle: Go to the Run menu of that Edit window, and select Run Module. Notice the shortcut (F5).

If the Shell window does not automatically come to the foreground, select it. You should see a line saying "RESTART" and then the start of the execution of the Mad Lib program with the cursor waiting for your entry after the first prompt. Finish executing the program. Be sure to type the final requested Enter, so you get back to the interpreter prompt: >>>

Look at the editor window again. You should see that different parts of the code have different colors. String literals are likely green. The reserved words `def` are likely orange. Look at the last two lines, where the identifier `tellStory` is black, and the identifier `input` is likely purple. Only identifiers that are not predefined by Python are black. If *you* create an identifier name, make sure Idle shows it in *black*.

1.9.2. A Bug Possible When Restarting Program Execution in Idle. When you execute a program from the Idle Editor, the interpreter gives a banner saying "RESTART", meaning that all the things you defined in any shell session so far are wiped clean and the program you are running starts fresh. There is one egregious exception to that, that was still present at least in the version of Idle for Python 3.1 in Windows. We will try to demonstrate the bug. (A *bug* is an error in a program.)

Start running the Mad Lib program again by going to the Editor Window containing madlib.py, and start running the program again, but do not continue....

You should see a prompt for user input generated by the program. Ignore this prompt and go back to the Edit Window and start the Mad Lib program again.

If this bug is still present, you should see a difference in this restart: This time after the RESTART banner and the interpreter prompt: >>>, which looks innocent enough, but this program should show the *program's* prompt string for input.

The problem only comes up because you interrupted the last execution when user input was being waited for. The restart was not complete here: The system is still looking for the pending user input from the last execution.

The fix is simple: Make sure the Interpreter Window is the currently selected window, and press return to terminate the lost user input. In some circumstances, you may need to press return a second time.

After that the program should start up normally with its prompt.

Watch out for this behavior, and remember the fix.

1.9.3. The Classic First Program . Make sure you have Idle started in your Python directory (in Windows with the provided Idle shortcut link), where you will store program files. (Do *not* start Idle from the Windows Start Menu!) If you just started Idle now, you may already have a blank Edit Window in front of you. If not, open a new window by going to the File menu and selecting New Window. This gives you a rather conventional text editing window with the mouse available, ability to cut and paste, plus a few special options for Python.

Type (or paste) the following into the *editor* window:

```
print('Hello world!')
```

Save the file with the File menu -> Save, and then enter the file name `hello.py`. Python program files should always be given a name ending in ".py", and you must enter the .py extension explicitly .

If you look in the editor, you should see that your text is color coded. The editor will color different parts of Python syntax in special colors. (In version 2.4 of Python, the coloring only happens after you save your file with the '.py' ending.)

Now that you have a complete, saved program, choose Run menu -> Run Module. You should see the program run in the Python Shell window.

You just wrote and executed a program. Unlike when you use the shell, this code is saved to a file in your Python folder. You can open and execute the file any time you want. (In Idle, use File->Open.)

To the interpreter, a program source file corresponds to a Python *module*. We will tend to use the more general term: a program file is a module. Note the term from the menu when running the program.

Distinguish program code from Shell text: It is easy to confuse the Shell and the Edit windows. Make sure you keep them straight. The hello.py program is just the line

```
print('Hello world!')
```

that you typed into the edit window and saved. When you ran the program in Idle, you saw results in the Shell. First came the Restart notice, the one-line output from the program saying hello, and a further Shell prompt:

```
>>> ================================ RESTART ========
>>>
Hello world!
>>>
```

You could also have typed this single printing line directly in the Shell in response to a Shell prompt. When you see `>>>`, you could enter the print function and get the exchange between you and the Shell:

```
>>> print('Hello world')
Hello world!
>>>
```

The three lines above are *not* a program you could save in a file and run. This is just an exchange in the *Shell*, with its `>>>` prompts, individual line to execute and the response. Again, just the single line, with no `>>>`,

```
print('Hello world!')
```

entered into the *Edit* window forms a program you can save and run. We will shortly get to more interesting many-statement programs, where it is much more convenient to use the Edit window than the Shell!

1.9.4. Program Documentation String. The program above is self evident, and shows how short and direct a program can be (unlike other languages like Java). Still, right away, get used to documenting a program. Python has a special feature: If the beginning of a program is just a quoted string, that string is taken to be the program's *documentation string*. Open the example file `hello2.py` in the Edit window:

```
'''A very simple program,
showing how short a Python program can be!
Authors: ___, ___
'''

print('Hello world!') #This is a stupid comment after the # mark
```

Most commonly, the initial documentation goes on for several lines, so a multi-line string delimiter is used (the triple quotes). Just for completeness of illustration in this program, another form of comment is also shown, a comment that starts with the symbol # and extends to the end of the line. The Python interpreter completely ignores this form of comment. Such a comment should only be included for better human understanding. Avoid making comments that do not really aid human understanding. (Do what I say, not what I did above.) Good introductory comment strings and appropriate names for the parts of your programs make fewer # symbol comments needed.

Run the program and see the documentation and comment make no difference in the result.

1.9.5. Screen Layout. Of course you can arrange the windows on your computer screen any way that you like. A suggestion as you start to use the combination of the editor to write, the shell to run, and the tutorial to follow along: Make all three mostly visible your computer screen at once. Drag the editor window to the upper left. Place the Shell window to the lower left, and perhaps reduce its height a bit so there is not much overlap. If you are looking at the web version of the tutorial on the screen, make it go top to bottom on the right, but not overlap the Idle windows too much. The web page rendering should generally adapt to the width pretty well. You can always temporarily maximize the window. Before resizing the browser window, it is good to look for an unusual phrase on your page, and search for it after resizing, since resizing can totally mess up your location in the web page.

There is an alternative to maximization for the Idle editor window: It you want it to go top to bottom of the screen but not widen, you can toggle that state with Alt-2. Play with all this.

1.10. Input and Output

1.10.1. The `input` Function. The hello program of Section 1.9.3 always does the same thing. This is not very interesting. Programs are only going to be reused if they can act on a variety of data. One way to get data is directly from the user. Modify the hello.py program as follows in the editor, and save it from the File menu with Save As...., using the name `hello_you.py`.

```
person = input('Enter your name: ')
print('Hello', person)
```

Run the program. In the Shell you should see

```
Enter your name:
```

Follow the instruction (and press Enter). Make sure the typing cursor is in the Shell window, at the end of this line. After you type your response, you can see that the program has taken in the line you typed. That is what the built-in function `input` does: First it prints the string you give as a parameter (in this case `'Enter your name: '`), and then it waits for a line to be typed in, and returns the string of characters you typed. In the `hello_you.py` program this value is assigned to the variable person, for use later.

The parameter inside the parentheses after `input` is important. It is a *prompt*, prompting you that keyboard input is expected at that point, and hopefully indicating what is being requested. Without the prompt, the user would not know what was happening, and the computer would just sit there waiting!

Open the example program, `interview.py`. Before running it (with any made-up data), see if you can figure out what it will do:

```
'''Illustrate input and print.'''

applicant = input("Enter the applicant's name: ")
interviewer = input("Enter the interviewer's name: ")
time = input("Enter the appointment time: ")
print(interviewer, "will interview", applicant, "at", time)
```

The statements are executed in the order they appear in the text of the program: *sequentially*. This is the simplest way for the execution of the program to flow. You will see instructions later that alter that natural flow.

If we want to reload and modify the `hello_you.py` program to put an exclamation point at the end, you could try:

```
person = input('Enter your name: ')
print('Hello', person, '!')
```

Run it and you see that it is not spaced right. There should be no space after the person's name, but the default behavior of the print function is to have each field printed separated by a space. There are several ways to fix this. You should know one. Think about it before going on to the next section. Hint: [4]

1.10.2. Print with Keyword Parameter sep. One way to put punctuation but no space after the person in hello_you.py is to use the plus operator, +. Another approach is to change the default separatpr between fields in the print function. This will introduce a new syntax feature, *keyword parameters*. The print function has a keyword parameter named sep. If you leave it out of a call to print, as we have so far, it is set equal to a space by default. If you add a final field, sep=”, in the print function in hello_you.py, you get the following example file, `hello_you2.py`:

```
person = input('Enter your name: ')
print('Hello ', person, '!', sep='')
```

Try the program.

Keyword paramaters must be listed at the end of the parameter list.

1.10.3. Numbers and Strings of Digits. Consider the following problem: Prompt the user for two numbers, and then print out a sentence stating the sum. For instance if the user entered 2 and 3, you would print “The sum of 2 and 3 is 5.”

You might imagine a solution like the example file `addition1.py`, shown below. There is a problem. Can you figure it out before you try it? Hint: [5] End up running it in any case.

```
x = input("Enter an integer: ")
y = input("Enter another integer: ")
print('The sum of ', x, ' and ', y, ' is ', x+y, '.', sep='')  # error!
```

[4]The + operation on strings adds no extra space.

[5]The input function produces values of string type.

We do not want string concatenation, but integer addition. We need integer operands. Briefly mentioned in Section 1.3 was the fact that we can use type names as functions to convert types. One approach would be to do that. Further variable names are also introduced in the example `addition2.py` file below to emphasize the distinctions in types. Read and run:

```
'''Conversion of strings to int before addition'''

xString = input("Enter an integer: ")
x = int(xString)
yString = input("Enter another integer: ")
y = int(yString)
print('The sum of ', x, ' and ', y, ' is ', x+y, '.', sep='')
```

Needing ito convert string input to numbersis a common situation, both with keyboard input and later in web pages. While the extra variables above emphasized the steps, it is more concise to write as in the variation in example file, `addition3.py`, doing the conversons to type `int` immediately:

```
'''Two numeric inputs'''

x = int(input("Enter an integer: "))
y = int(input("Enter another integer: "))
print('The sum of ', x, ' and ', y, ' is ', x+y, '.', sep='')
```

The simple programs so far have followed a basic *programming pattern*: input-calculate-output. Get all the data first, calculate with it second, and output the results last. The pattern sequence would be even clearer if we explicitly create a named result variable in the middle, as in `addition4.py`:

```
x = int(input("Enter an integer: "))
y = int(input("Enter another integer: "))
sum = x + y
print('The sum of ', x, ' and ', y, ' is ', sum, '.', sep='')
```

We will see more complicated patterns, which involve repetition, in the future.

EXERCISE 1.10.3.1. * Write a version, `add3.py`, that asks for three numbers, and lists all three, and their sum, in similar format to the example above.

EXERCISE 1.10.3.2. * a. Write a program, `quotient.py`, that prompts the user for two integers, and then prints them out in a sentence with an integer division problem like `"The quotient of 14 and 3 is 4 with a remainder of 2"`. Review Section 1.4.3 if you forget the integer division or remainder operator.

1.10.4. String Format Operation.

A common convention is fill-in-the blanks. For instance,

```
Hello _____!
```

and you can fill in the name of the person greeted, and combine given text with a chosen insertion. Python has a similar construction, better called fill-in-the-braces. There is a particular operation on strings called `format`, that makes substitutions into places enclosed in braces. For instance the example file, `hello_you3.py`, creates and prints the same string as in `hello_you2.py` from the previous section:

```
person = input('Enter your name: ')
greeting = 'Hello {}!'.format(person)
print(greeting)
```

There are several new ideas here!.

First *method* calling syntax is used. You will see in this more detail at the beginning of the next chapter. Strings and other objects have a special syntax for functions, called *methods*, associated with the *particular type of object*. In particular `str` objects have a method called `format`. The syntax for methods has the object followed by a period followed by the method name, and further parameters in parentheses.

object.methodname(paramters)

In the example above, the object is the string `'Hello {}!'`. The method is named `format`. There is one further parameter, `person`.

The string has a special form, with braces embedded. Places where braces are embedded are replaced by the value of an expression taken from the parameter list for the format method. There are many variations

on the syntax between the braces. In this case we use the syntax where the first (and only) location in the string with braces has a substitution made from the first (and only) parameter

In the code above, this new string is assigned to the identifier `greeting`, and then the string is printed. The identifier `greeting` was introduced to break the operations into a clearer sequence of steps. Since the value of `greeting` is only referenced once, it can be eliminated with the more concise version:

```
person = input('Enter your name: ')
print('Hello {}!'.format(person))
```

Consider the interview program. Suppose we want to add a period at the end of the sentence (with no space before it). One approach would be to combine everything with plus signs. Another way is printing with keyword `sep=''`. Another approach is with string formating. Here the idea is to fill in the blanks in

```
_____ will interview _____ at _____.
```

There are multiple places to substiitute, and the format approach can be extended to multiple substitutions: Each place in the format string where there is `'{}'`, the `format` operation will substitute the value of the next parameter in the format parameter list.

Run the example file `interview2.py`, and check that the results from all three methods match.

```
'''Compare different approaches to printing with embedded values.'''

applicant = input("Enter the applicant's name: ")
interviewer = input("Enter the interviewer's name: ")
time = input("Enter the appointment time: ")
print(interviewer + ' will interview ' + applicant + ' at ' + time +'.')
print(interviewer, ' will interview ', applicant, ' at ', time, '.', sep='')
print('{} will interview {} at {}.'.format(interviewer, applicant, time))
```

A technical point: Since braces have special meaning in a format string, there must be a special rule if you want braces to actually be included in the final formatted string. The rule is to double the braces: `'{{'` and `'}}'`. The example code `formatBraces.py`, shown below, makes `setStr` refer to the string `'The set is {5, 9}.'`. The initial and final doubled braces in the format string generate literal braces in the formatted string:

```
a = 5
b = 9
formatStr = 'The set is {{{}, {}}}.'
setStr = formatStr.format(a, b)
print(setStr)
```

This kind of format string depends directly on the order of the parameters to the format method. There is another approach with a dictionary, that was used in the first sample program, and will be discussed more in Section 1.12.2 on dictionaries. The dictionary approach is probably the best in many cases, but the count-based approach is an easier start, particularly if the parameters are just used once, in order.

(Optional elaboration) Imagine the format parmaters numbered in order, starting from 0. In this case 0, 1, and 2. The number of the parameter position may be included inside the braces, so an alternative to the last line of `interview2.py` is (added in example file `interview3.py`):

```
print('{0} will interview {1} at {2}.'.format(interviewer, applicant, time))
```

This is more verbose than the previous version, with no obvious advantage. If you desire to use some of the parameters more than once, then the approach with the numerical identification with the parameters is useful. Every place the string includes `'{0}'`, the `format` operation will substitute the value of the initial parameter in the list. Wherever `'{1}'` appears, the next format parameter will be substituted....

Predict the results of the example file `arith.py` shown below,and then check yourself by running it. In this case the numbers referring to the parameter positions are necessary. They are both repeated and used out of order:

```
'''Fancier format string example.'''
```

```
x = 20
y = 30
formatStr = '{0} + {1} = {2}; {0} * {1} = {3}.'
equations = formatStr.format(x, y, x+y, x*y)
print(equations)
```

Try the program.

EXERCISE 1.10.4.1. * Write a version of Exercise 1.10.3.1, `add3f.py`, that uses the string format method to construct the final string.

EXERCISE 1.10.4.2. * Write a version of Exercise 1.10.3.2, `quotientformat.py`, that uses the string format method to construct the final string.

1.11. Defining Functions of your Own

1.11.1. Syntax Template Typography. When new Python syntax is introduced, the usual approach will be to give both specific examples and general templates. In general templates for Python syntax the typeface indicates the the category of each part:

Typeface	Meaning
`Typewriter font`	Text to be written verbatim
Emphasized	A place where you can use an arbitrary identifier. The emphasized text attempts to be descriptive of the meaning of the identifier in the current context.
Normal text	A description of what goes in that position, without giving explicit syntax

We will use these conventions shortly in the discussion of function syntax, and will continue to use the conventions throughout the tutorial.

1.11.2. A First Function Definition. If you know it is the birthday of a friend, Emily, you might tell those gathered with you to sing "Happy Birthday to Emily".

We can make Python display the song. *Read*, and run if you like, the example program `birthday1.py`:

```
print("Happy Birthday to you!")
print("Happy Birthday to you!")
print("Happy Birthday, dear Emily.")
print("Happy Birthday to you!")
```

You would probably not repeat the whole song to let others know what to sing. You would give a request to sing via a descriptive name like "Happy Birthday to Emily".

In Python we can also give a name like `happyBirthdayEmily`, and associate the name with whole song by using a *function definition*. We use the Python `def` keyword, short for *define*.

Read for now:

```
def happyBirthdayEmily():
    print("Happy Birthday to you!")
    print("Happy Birthday to you!")
    print("Happy Birthday, dear Emily.")
    print("Happy Birthday to you!")
```

There are several parts of the syntax for a function definition to notice:

The *heading* contains `def`, the name of the function, parentheses, and finally a colon.

`def` *`function_name`*`():`

The remaining lines form the function *body* and are indented by a consistent amount. (The exact amount is not important to the interpreter, though 2 or 4 spaces are common conventions.)

The whole definition does just that: *defines* the meaning of the name `happyBirthdayEmily`, but it does not do anything else yet – for example, the definition itself does not make anything be printed yet. This is our first example of altering the order of execution of statements from the normal sequential order. This is important: the statements in the function *definition* are *not* executed as Python first passes over the lines.

The code above is in example file `birthday2.py`. Load it in Idle and execute it from there. *Nothing* should happen visibly. This is just like defining a variable: Python just remembers the function definition

for future reference. After Idle finished executing a program, however, its version of the Shell remembers function definitions from the program.

In the Idle *Shell* (not the editor), enter

```
happyBirthdayEmily
```

The result probably surprises you! When you give the Shell an identifier, it tells you its *value*. Above, without parentheses, it identifies the function code as the value (and gives a location in memory of the code). Now try the name in the Idle Shell with *parentheses* added:

```
happyBirthdayEmily()
```

The parentheses tell Python to *execute* the named function rather than just *refer* to the function. Python goes back and looks up the definition, and only then, executes the code inside the function definition. The term for this action is a *function call* or function *invocation*. Note, in the function *call* there is no `def`, but there is the function name followed by parentheses.

`function_name``()`

In many cases we will use a feature of program execution in Idle: that after program execution is completed, the Idle Shell still remembers functions defined in the program. This is not true if you run a program by selecting it directly in the operating system. *The general assumption in this Tutorial will be that programs are run in Idle and the Idle Shell is the Shell referred to.* It will be explicitly stated when you should run a program directly from the operating system. (With most of the examples in the tutorial, running from the operating system is OK – the execution method will not actually matter.)

Look at the example program `birthday3.py`. See it just adds two more lines, *not* indented. Can you guess what it does? Try it:

```
def happyBirthdayEmily():                #1
    print("Happy Birthday to you!")      #2
    print("Happy Birthday to you!")      #3
    print("Happy Birthday, dear Emily.") #4
    print("Happy Birthday to you!")      #5

happyBirthdayEmily()                     #6
happyBirthdayEmily()                     #7
```

The *execution* sequence is different from the *textual* sequence:

(1) Lines 1-5: Python starts from the top, reading and remembering the definition. The definition ends where the indentation ends. (The code also shows a blank line there, but that is only for humans, to emphasize the end of the definition.)
(2) Line 6: this is not indented inside any definition, so the interpreter executes it directly, calling `happyBirthdayEmily()` while remembering where to return.
(3) Lines 1-5: The code of the function is executed for the first time, printing out the song.
(4) End of line 6: Back from the function call. continue on.
(5) Line 7: the function is called again while this location is remembered.
(6) Lines 1-5: The function is executed again, printing out the song again.
(7) End of line 7: Back from the function call, but at this point there is nothing more in the program, and execution stops.

Functions alter execution order in several ways: by statements not being executed as the definition is first read, and then when the function is called during execution, jumping to the function code, and back at the the end of the function execution.

If it also happens to be Andre's birthday, we might define a function `happyBirthdayAndre`, too. Think how to do that before going on ....

1.11.3. Multiple Function Definitions. Here is example program `birthday4.py` where we add a function `happyBirthdayAndre`, and call them both. Guess what happens, and then try it:

```
def happyBirthdayEmily():  # same old function
    print("Happy Birthday to you!")
    print("Happy Birthday to you!")
```

```
    print("Happy Birthday, dear Emily.")
    print("Happy Birthday to you!")

def happyBirthdayAndre():
    print("Happy Birthday to you!")
    print("Happy Birthday to you!")
    print("Happy Birthday, dear Andre.")
    print("Happy Birthday to you!")

happyBirthdayEmily()
happyBirthdayAndre()
```

Again, everything is definitions except the last two lines. They are the only lines executed directly. The calls to the functions *happen* to be in the same order as their definitions, but that is arbitrary. If the last two lines were swapped, the order of operations would change. Do swap the last two lines so they appear as below, and see what happens when you execute the program:

```
happyBirthdayAndre()
happyBirthdayEmily()
```

Functions that you write can also call other functions you write. It is a good convention to have the main action of a program be in a function for easy reference. The example program `birthday5.py` has the two Happy Birthday calls inside a final function, `main`. Do you see that this version accomplishes the same thing as the last version? Run it.

```
def happyBirthdayEmily():                 #1
    print("Happy Birthday to you!")       #2
    print("Happy Birthday to you!")       #3
    print("Happy Birthday, dear Emily.")  #4
    print("Happy Birthday to you!")       #5

def happyBirthdayAndre():                 #6
    print("Happy Birthday to you!")       #7
    print("Happy Birthday to you!")       #8
    print("Happy Birthday, dear Andre.")  #9
    print("Happy Birthday to you!")      #10

def main():                              #11
    happyBirthdayAndre()                 #12
    happyBirthdayEmily()                 #13

main()                                   #14
```

If we want the program to do anything automatically when it is runs, we need one line outside of definitions! The final line is the only one directly executed, and it calls the code in `main`, which in turn calls the code in the other two functions.

Detailed order of execution:

(1) Lines 1-13: Definitions are read and remembered
(2) Line 14: The only line outside definitions, is executed directly. This location is remembered as main is executed.
(3) Line 11: Start on main
(4) Line 12. This location is remembered as execution jumps to happyBirthdayAndre
(5) Lines 6-10 are executed and Andre is sung to.
(6) Return to the end of Line 12: Back from happyBirthdayAndre function call
(7) Line 13: Now happyBirthdayEmily is called as this location is remembered.
(8) Lines 1-5: Sing to Emily
(9) Return to the end of line 13: Back from happyBirthdayEmily function call, done with main
(10) Return to the end of line 14: Back from main; at the end of the program

There is one practical difference from the previous version. After execution, if we want to give another round of Happy Birthday to *both* persons, we only need to enter one further call in the *Shell* to:

```
main()
```

As a simple example emphasizing the significance of a line being indented, guess what the the example file `order.py` does, and run it to check:

```
def f():
    print('In function f')
    print('When does this print?')
f()
```

Modify the file so the second print function is *out*dented like below. What should happen now? Try it:

```
def f():
    print('In function f')
print('When does this print?')
f()
```

The lines indented inside the function definition are *remembered* first, and only executed when the function f is invoked at the end. The lines outside any function definition (not indented) are executed in order of appearance.

Exercise 1.11.3.1. * Write a program, `poem.py`, that defines a function that prints a *short* poem or song verse. Give a meaningful name to the function. Have the program end by calling the function three times, so the poem or verse is repeated three times.

1.11.4. Function Parameters. As a young child, you probably heard Happy Birthday sung to a couple of people, and then you could sing to a new person, say Maria, without needing to hear the whole special version with Maria's name in it word for word. You had the power of *abstraction*. With examples like the versions for Emily and Andre, you could figure out what change to make it so the song could be sung to Maria!

Unfortunately, Python is not that smart. It needs explicit rules. If you needed to explain *explicitly* to someone how Happy Birthday worked in general, rather than just by example, you might say something like this:

First you have to be *given* a person's name. Then you sing the song with the person's name inserted at the end of the third line.

Python works something like that, but with its own syntax. The term "person's name" serves as a stand-in for the actual data that will be used, "Emily", "Andre", or "Maria". This is just like the association with a variable name in Python. "person's name" is not a legal Python identifier, so we will use just `person` as this stand-in.

The function definition indicates that the variable name `person` will be used inside the function by inserting it between the parentheses of the definition. Then in the body of the definition of the function, person is used in place of the real data for any specific person's name. Read and then run example program `birthday6.py`:

```
def happyBirthday(person):                      #1
    print("Happy Birthday to you!")             #2
    print("Happy Birthday to you!")             #3
    print("Happy Birthday, dear " + person + ".") #4
    print("Happy Birthday to you!")             #5

happyBirthday('Emily')                          #6
happyBirthday('Andre')                          #7
```

In the definition heading for `happyBirthday`, `person` is referred to as a *parameter*, or a *formal parameter*. This variable name is a placeholder for the real name of the person being sung to.

The last two lines of the program, again, are the only ones outside of definitions, so they are the only ones executed directly. There is now an actual name between the parentheses in the function calls. The value between the parentheses here in the function call is referred to as an *argument* or *actual parameter* of the function call. The argument supplies the actual data to be used in the function execution. When the call is

made, Python does this by associating the formal parameter name `person` with the actual parameter data, as in an assignment statement. In the first call, this actual data is `'Emily'`. We say the actual parameter value is *passed* to the function.

The execution in greater detail:

(1) Lines 1-5: Definition remembered
(2) Line 6: Call to `happyBirthday`, with actual parameter `'Emily'`.
(3) Line 1: `'Emily'` is passed to the function, so `person = 'Emily'`
(4) Lines 2-5: The song is printed, with `'Emily'` used as the value of `person` in line 4: printing `'Happy birthday, dear Emily.'`
(5) End of line 6: Return from the function call and continue
(6) Line 7: Call to `happyBirthday`, this time with actual parameter `'Andre'`
(7) Line 1: `'Andre'` is passed to the function, so `person = 'Andre'`
(8) Lines 2-5: The song is printed, with `'Andre'` used as the value of `person` in line 4: printing `'Happy birthday, dear Andre.'`
(9) End of line 7: Return from the function call, and the program is over.

The beauty of this system is that the same function definition can be used for a call with a different actual parameter variable, and then have a different effect. The value of the variable person is used in the third line of `happyBirthday`, to put in whatever actual parameter value was given.

This is the power of *abstraction*. It is one application of the most important principal in programming. Rather than have a number of separately coded parts with only slight variations, see where it is appropriate to combine them using a function whose parameters refer to the parts that are different in different situations. Then the code is written to be simultaneously appropriate for the separate specific situations, with the substitutions of the right parameter values.

You can go back to having a main function again, and everything works. Run `birthday7.py`:

```
def happyBirthday(person):
    print("Happy Birthday to you!")
    print("Happy Birthday to you!")
    print("Happy Birthday, dear " + person + ".")
    print("Happy Birthday to you!")

def main():
    happyBirthday('Emily')
    happyBirthday('Andre')

main()
```

EXERCISE 1.11.4.1. * Make your own further change to the file and save it as `birthdayMany.py`: Add a function call, so Maria gets a verse, in addition to Emily and Andre. Also print a blank line between verses. (You may either do this by adding a print line to the function definition, or by adding a print line between all calls to the function.)

We can combine function parameters with user input, and have the program be able to print Happy Birthday for anyone. Check out the main method and run `birthday_who.py`:

```
def happyBirthday(person):
    print("Happy Birthday to you!")
    print("Happy Birthday to you!")
    print("Happy Birthday, dear " + person + ".")
    print("Happy Birthday to you!")

def main():
    userName = input("Enter the Birthday person's name: ")
    happyBirthday(userName)

main()
```

This last version illustrates several important ideas:

(1) There are more than one way to get information into a function:
    (a) Have a value passed in through a parameter.
    (b) Prompt the user, and obtain data from the keyboard.
(2) It is a good idea to separate the *internal* processing of data from the *external* input from the user by the use of distinct functions. Here the user interaction is in `main`, and the data is manipulated in `happyBirthday`.
(3) In the first examples of actual parameters, we used literal values. In general an actual parameter can be an expression. The expression is evaluated before it is passed in the function call. One of the simplest expressions is a plain variable name, which is evaluated by replacing it with its associated value. Since it is only the value of the actual parameter that is passed, not any variable name, there is no need to have an actual parameter variable name match a formal parameter name. (Here we have the value of `userName` in `main` becoming the value of `person` in `happyBirthday`.)

1.11.5. Multiple Function Parameters. A function can have more than one parameter in a parameter list separated by commas. Here the example program `addition5.py` uses a function to make it easy to display many sum problems. Read and follow the code, and then run:

```
def sumProblem(x, y):
    sum = x + y
    print('The sum of ', x, ' and ', y, ' is ', sum, '.', sep='')

def main():
    sumProblem(2, 3)
    sumProblem(1234567890123, 535790269358)
    a = int(input("Enter an integer: "))
    b = int(input("Enter another integer: "))
    sumProblem(a, b)

main()
```

The actual parameters in the function call are evaluated left to right, and then these values are associated with the formal parameter names in the function definition, also left to right. For example the function call with actual parameters, `f(actual1, actual2, actual3)`, calling the function f with definition heading
`def f(formal1, formal2, formal3):`
acts approximately as if the first lines executed inside the called function were

```
formal1 = actual1
formal2 = actual2
formal3 = actual3
```

Functions provide extremely important functionality to programs, allowing task to be defined once and performed repeatedly with different data. It is essential to see the difference between the formal parameters used to describe what is done inside the function definition (like x and y in the definition of sumProblem) and the actual parameters (like 2 and 3 or 1234567890123 and 535790269358) which *substitute* for the formal parameters when the function is actually executed. The main method above uses three different sets of actual parameters in the three calls to sumProblem.

EXERCISE 1.11.5.1. '* Modify the program above and save it as `quotientProb.py`. The new program should have a quotientProblem function, printing as in the Exercise 1.10.3.2. The main method should test the function on several sets of literal values, and also test the function with input from the user.

1.11.6. Returned Function Values. You probably have used mathematical functions in algebra class, but they all had calculated values associated with them. For instance if you defined $f(x) = x^2$, then it follows that $f(3)$ is $3^2 = 9$, and $f(3)+f(4)$ is $3^2+4^2 = 25$. Function calls in expressions get replaced during evaluation by the value of the function.

The corresponding definition and examples in Python would be the following, also in the example program `return1.py`. *Read and run*:

```
def f(x):
    return x*x

print(f(3))
print(f(3) + f(4))
```

The new Python syntax is the *return statement*, with the word `return` followed by an expression. Functions that return values can be used in expressions, just like in math class. When an expression with a function call is evaluated, the function call is effectively replaced temporarily by its returned value. Inside the Python function, the value to be returned is given by the expression in the `return` statement. After the function `f` finishes executing from inside

```
print(f(3))
```

it is as if the statement temporarily became

```
print(9)
```

and similarly when executing

```
print(f(3) + f(4))
```

the interpreter first evaluates f(3) and effectively replaces the call by the returned result, 9, as if the statement temporarily became

```
print(9 + f(4))
```

and then the interpreter evaluates f(4) and effectively replaces the call by the returned result, 16, as if the statement temporarily became

```
print(9 + 16)
```

resulting finally in 25 being calculated and printed.

Python functions can return any type of data, not just numbers, and there can be any number of statements executed before the return statement. Read, follow, and run the example program `return2.py`:

```
def lastFirst(firstName, lastName):               #1
    separator = ', '                              #2
    result = lastName + separator + firstName     #3
    return result                                 #4

print(lastFirst('Benjamin', 'Franklin'))          #5
print(lastFirst('Andrew', 'Harrington'))          #6
```

The code above has a new feature, variables `separator` and `result` are given a value in the function, but `separator` and `result` are not among the formal parameters. The assignments work as you would expect here. More on this shortly, in Section 1.11.8 on local scope.

Details of the execution:

(1) Lines 1-4: Remember the definition
(2) Line 5: call the function, remembering where to return
(3) Line 1: pass the parameters: `firstName = 'Benjamin'`; `lastName = 'Franklin'`
(4) Line 2: Assign the variable `separator` the value `', '`
(5) Line 3: Assign the variable `result` the value of
`lastName + separator + firstName` which is
`'Franklin' + ', ' + 'Benjamin'`, which evaluates to
`'Franklin, Benjamin'`
(6) Line 4: Return `'Franklin, Benjamin'`
(7) Line 5 Use the value returned from the function call so the line effectively becomes
`print('Franklin, Benjamin')`
so print it.
(8) Line 6: call the function with the new actual parameters, remembering where to return
(9) Line 1: pass the parameters: `firstName = 'Andrew'`; `lastName = 'Harrington'`
(10) Lines 2-4: ... calculate and return `'Harrington, Andrew'`
(11) Line 6: Use the value returned by the function and print `'Harrington, Andrew'`

Compare `return2.py` and `addition5.py`, from the previous section. Both use functions. Both print, but where the printing is done differs. The function `sumProblem` prints directly inside the function and returns nothing explicitly. On the other hand `lastFirst` does not print anything but returns a string. The caller gets to decide what to do with the string, and above it is printed in the main program.

Open `addition5.py` again, and introduce a *common mistake*. Change the last line of the function `main` inserting `print`, so it says

```
print(sumProblem(a, b))
```

Then try running the program. The desired printing is actually done inside the function sumProblem. You introduced a statement to print what `sumProblem` *returns*. Although `sumProblem` returns nothing *explicitly*, Python does make every function return something. If there is nothing explicitly returned, the special value `None` is returned. You should see that in the Shell output. This is a fairly common error. If you see a 'None' is your output where you do not expect it, it is likely that you have printed the return value of a function that did not return anything explicitly!

Exercise 1.11.6.1. Create quotientReturn.py by modifying quotientProb.py from Exercise 1.11.5.1 so that the program accomplishes the same thing, but everywhere change the quotientProblem function into one called `quotientString` that merely *returns* the string rather than printing the string directly. Have the `main` function print the result of each call to the quotientString function.

1.11.7. Two Roles: Writer and Consumer of Functions. The remainder of Section1.11 covers finer points about functions that you might skip on a first reading.

We are only doing tiny examples so far to get the basic idea of functions. In much larger programs, functions are useful to manage complexity, splitting things up into logically related, modest sized pieces. Programmers are both writers of functions and consumers of the other functions called inside their functions. It is useful to keep those two roles separate:

The user of an already written function needs to know:

(1) the name of the function
(2) the order and meaning of parameters
(3) what is returned or produced by the function

How this is accomplished is not relevant at this point. For instance, you use the work of the Python development team, calling functions that are built into the language. You need know the three facts about the functions you call. You do not need to know exactly *how* the function accomplishes its purpose.

On the other hand when you *write* a function you need to figure out exactly how to accomplish your goal, name relevant variables, and write your code, which brings us to the next section.

1.11.8. Local Scope. For the logic of writing functions, it is important that the writer of a function knows the names of variables inside the function. On the other hand, if you are only using a function, maybe written by someone unknown to you, you should not care what names are given to values used internally in the implementation of the function you are calling. Python enforces this idea with *local scope* rules: Variable names initialized and used inside one function are *invisible* to other functions. Such variables are called *local* variables. For example, an elaboration of the earlier program `return2.py` might have its `lastFirst` function with its local variable `separator`, but it might also have another function that defines a `separator` variable, maybe with a different value like `'\n'`. They do not conflict. They are independent. This avoids lots of errors!

For example, the following code in the example program `badScope.py` causes an execution error. Read it and run it, and see:

```
def main():
    x = 3
    f()

def f():
    print(x)  #f does not know about the x defined in main

main()
```

We will fix this error below. The execution error message mentions "global name". Names defined outside any function definition, at the "top-level" of your program are called *global*. They are a special case. They are discussed more in the next section.

If you do want local data from one function to go to another, define the called function so it includes parameters! Read and compare and try the program `goodScope.py`:

```
def main():
    x = 3
    f(x)

def f(x):
    print(x)

main()
```

With parameter passing, the parameter name `x` in the function `f` does not need to match the name of the actual parameter in `main`. The definition of `f` could just as well have been:

```
def f(whatever):
    print(whatever)
```

1.11.9. Global Constants. If you define global variables (outside of any function definition), they are visible inside all of your functions. It is good programming practice to avoid defining global variables and instead to put your variables inside functions and explicitly pass them as parameters where needed. One common exception is constants: A *constant* is a name that you give a fixed data value to, by assigning a value to the name only in a single assignment statement. You can then use the name of the fixed data value in expressions later. A simple example program is `constant.py`:

```
PI = 3.14159265358979   # global constant -- only place the value of PI is set

def circleArea(radius):
    return PI*radius*radius    # use value of global constant PI

def circleCircumference(radius):
    return 2*PI*radius         # use value of global constant PI

print('circle area with radius 5:', circleArea(5))
print('circumference with radius 5:', circleCircumference(5))
```

This example uses numbers with decimal points, discussed more in Section 1.14.1. By convnetion, names for constants are all capital letters.

Issues with global variables do not come up if they are only used as constants.

Function names defined at the top-level also have global scope. This is what allows you to use one function you defined inside another function you define.

1.12. Dictionaries

1.12.1. Definition and Use of Dictionaries. In common usage, a dictionary is a collection of words matched with their definitions. Given a word, you can look up its definition. Python has a built in dictionary type called dict which you can use to create dictionaries with arbitrary definitions for character strings. It can be used for the common usage, as in a simple English-Spanish dictionary.

Look at the example program `spanish1.py` and run it.

```
"""A tiny English to Spanish dictionary is created,
using the Python dictionary type dict.
Then the dictionary is used, briefly.
"""

spanish = dict()
```

```
spanish['hello'] = 'hola'
spanish['yes'] = 'si'
spanish['one'] = 'uno'
spanish['two'] = 'dos'
spanish['three'] = 'tres'
spanish['red'] = 'rojo'
spanish['black'] = 'negro'
spanish['green'] = 'verde'
spanish['blue'] = 'azul'

print(spanish['two'])
print(spanish['red'])
```

First an empty dictionary is created using `dict()`, and it is assigned the descriptive name `spanish`.

To refer to the definition for a word, you use the dictionary name, follow it by the word inside *square brackets*. This notation can either be used on the left-hand side of an assignment to make (or remake) a definition, or it can be used in an expression (as in the print functions), where its definition is one stored earlier into the dictionary. For example,

`spanish['hello'] = 'hola'`

makes an entry in our `spanish` dictionary for `'hello'` , where the definition matched to it is `'hola'`.

`print(spanish['red'])`

retrieves the definition for `'red'`, which is `'rojo'`.

Since the Spanish dictionary is defined at the top-level, the variable name `spanish` is still defined after the program runs: after running the program, use `spanish` in the Shell to check out the translations of some more words, other than `'two'` and `'red'`.

Creating the dictionary is quite a different activity from the use at the end of the code, so with functions to encapsulate the tasks, we could write the example program `spanish2.py` instead, with the same result:

```
"""A tiny English to Spanish dictionary is created,
using the Python dictionary type dict.
Then the dictionary is used, briefly.
"""

def createDictionary():
    '''Returns a tiny Spanish dictionary'''
    spanish = dict() # creates an empty dictionary
    spanish['hello'] = 'hola'
    spanish['yes'] = 'si'
    spanish['one'] = 'uno'
    spanish['two'] = 'dos'
    spanish['three'] = 'tres'
    spanish['red'] = 'rojo'
    spanish['black'] = 'negro'
    spanish['green'] = 'verde'
    spanish['blue'] = 'azul'
    return spanish

def main():
    dictionary = createDictionary()
    print(dictionary['two'])
    print(dictionary['red'])

main()
```

This code illustrates several things about functions.

- First, like whole files, functions can have a documentation string immediately after the definition heading. It is a good idea to document the return value!
- The dictionary that is created is returned, but the local variable name in the function, `spanish`, is lost when the function terminates.
- In `main`, to remember the dictionary returned, it needs a name. The name does not have to match the name used in `createDictionary`. The name `dictionary` is descriptive.

We could also use the dictionary more extensively. The example program spanish2a.py is the same as above except it has the following main method

```
def main():
    dictionary = createDictionary()
    print('Count in Spanish: ' + dictionary['one'] + ', ' +
          dictionary['two'] + ', ' + dictionary['three'] + ',...')
    print('Spanish colors: ' + dictionary['red'] + ', ' +
          dictionary['blue'] + ', ' + dictionary['green'] + ',...')
```

Try it, and check that it makes sense.

Python dictionaries are actually more general than the common use of dictionaries. They do not have to associate words and their string definitions. They can associate many types of objects with some arbitrary object. The more general Python terminology for word and definition are *key* and *value*. Given a key, you can look up the corresponding value. The only restriction on the key is that it be an *immutable* type. This means that a value of the key's type cannot be changed internally after it is initially created. Strings and numbers are immutable. A dictionary is *mutable*: its value can be changed internally. (You can add new definitions to it!) We will see more mutable and immutable types later and explore more of the internal workings of data types.

Exercise 1.12.1.1. * Write a tiny Python program `numDict.py` that makes a dictionary whose keys are the words 'one', 'two', 'three', and 'four', and whose corresponding values are the numerical equivalents, 1, 2, 3, and 4 (ints, not strings). Include code to test the resulting dictionary by referencing several of the definitions and printing the results.

1.12.2. Dictionaries and String Formatting. At the end of the main function in `spanish2a.py` from the last section, two strings are constructed and printed. The expressions for the two strings include a sequence of literal strings concatenated with interspersed values from a dictionary. There is a much neater, more readable way to generate these strings. We will develop this in several steps. The first string could be constructed and printed as follows:

```
numberFormat = "Count in Spanish: {one}, {two}, {three}, ..."
withSubstitutions = numberFormat.format(one='uno', two='dos', three='tres')
print(withSubstitutions)
```

There are several new ideas here!.

Note the form of the string assigned the name `numberFormat`: It has the English words for numbers in *braces* where we want the Spanish definitions substituted.

The second line uses *method* calling syntax. You will see this in more detail at the beginning of the next chapter. Strings and other objects have a special syntax for functions tightly associated with the particular type of object. Such functions are called methods. In particular str objects have a method called `format`. The syntax for methods

object.methodname(paramters)

has the object followed by a period followed by the method name, and further parameters in parentheses.

In the example above, the object is the string called `numberFormat`. The method is named `format`. The parameters in this case are all *keyword* parameters. You have already seen keyword parameters `sep` and `end` used in print function calls. In this particular application, the keywords are chosen to include all the words that appear enclosed in braces in the `numberFormat` string.

When the string `numberFormat` has the `format` method applied to it with the given keyword parameters, a new string is created with substitutions into the places enclosed in braces. The substitutions are just the values given by the keyword parameters. Hence the printed result is

```
Count in Spanish: uno, dos, tres, ...
```

Now we go one step further: The keyword parameters associate the keyword names with the values after the equal signs. The dictionary from `spanish2a.py` includes exactly the same associations. There is a special notation allowing such a dictionary to supply keyword parameters. Assuming `dictionary` is the Spanish dictionary from `spanish2a.py`, the method call

```
numberFormat.format(one='uno', two='dos', three='tres')
```

returns the same string as

```
numberFormat.format(**dictionary)
```

The special syntax ** before the dictionary indicates that the dictionary is not to be treated as a single regular parameter. Instead keyword arguments for all the entries in the dictionary effectively appear in its place.

Below is a substitute for the main method in `spanish2a.py`. The whole revised program is in example program `spanish3.py`.

```
def main():
    dictionary = createDictionary()
    numberFormat = "Count in Spanish: {one}, {two}, {three}, ..."
    withSubstitutions = numberFormat.format(**dictionary)
    print(withSubstitutions)
    print("Spanish colors: {red}, {blue}, {green}, ...".format(**dictionary))
```

The string with the numbers is constructed in steps as discussed above. The printing of the string with the Spanish colors is coded more concisely. There are not named variables for the format string or the resulting formatted string. You are free to use either coding approach.

In general, use this syntax for the string format method with a dictionary, returning a new formatted string:

formatString`.format(**`*aDictionary*`)`

where the format string contains dictionary keys in braces where you want the dictionary values substituted. The dictionary key names must follow the rules for legal identifiers.

At this point we have discussed in some detail everything that went into the first sample program, `madlib.py`, of Section 1.2.3! This is certainly the most substantial program so far.

Look at `madlib.py` again, see how we have used most of the ideas so far. If you want more description, you might look at section 1.2.3 again (or for the first time): it should make much more sense now.

Exercise 1.12.2.1. To confirm your better understanding of `madlib.py`, load it in the editor, *rename* it as myMadlib.py, and *modify it to have a less lame story*, with more and different entries in the dictionary. Make sure addPick is called for each key in your format string. Test your version.

We will use `madlib.py` as a basis for more substantial modifications in structure in Section 2.3.3.

1.12.3. Dictionaries and Python Variables. Dictionaries are central to the implementation of Python. Each variable identifier is associated with a particular value. These relationships are stored in dictionaries in Python, and these dictionaries are accessible to the user: You can use the function call `locals()` to return a dictionary containing all the current local variables names as keys and all their values as the corresponding dictionary values. This dictionary can be used with the string format method, so you can embed local variable names in a format string and use then very easily!

For example, run the example program `arithDict.py`:

```
'''Fancier format string example, with locals().'''

x = 20
y = 30
sum = x+y
prod = x*y
formatStr = '{x} + {y} = {sum}; {x} * {y} = {prod}.'
equations = formatStr.format(**locals())
print(equations)
```

Note the variable names inside braces in `formatStr,` and the dictionary reference used as the format parameter is `**locals()`.

A string like `formatStr` is probably the most readable way to code the creation of a string from a collection of literal strings and program values. The ending part of the syntax, `.format(**locals())`, may appear a bit strange, but it is very useful! We will use this notation extensively to clearly indicate how values are embedded into strings.

The example program `hello_you4.py` does the same thing as the earlier hello_you versions, but with a dictionary reference:

```
person = input('Enter your name: ')
greeting = 'Hello {person}!'.format(**locals())
print(greeting)
```

1.13. Loops and Sequences

Modern computers can do millions or even billions of instructions a second. With the techniques discussed so far, it would be hard to get a program that would run by itself for more than a fraction of a second.[6] Practically, we cannot write millions of instructions to keep the computer busy. To keep a computer doing useful work we need *repetition*, looping back over the same block of code again and again. There are two Python statement types to do that: the simpler `for` loops, which we take up shortly, and `while` loops, which we take up later, in Section 3.3. Two preliminaries: First, the value of already defined variables can be updated. This will be particularly important in loops. We start by following how variables can be updated in an even simpler situation. Second, *for* loops involve sequence types, so we will first look at a basic sequence type: `list`. This is a long section. Go carefully.

1.13.1. Updating Variables. The programs so far have defined and used variables, but other than in early shell examples we have not changed the value of existing variables. For now consider a particularly simple example, just chosen as an illustration, in the example file `updateVar.py`:

```
x = 3           #1
y = x + 2       #2
y = 2*y         #3
x = y - x       #4
print(x, y)     #5
```

Can you *predict* the result? Run the program and check. Particularly if you did not guess right, it is important to understand what happens, one step at a time. That means keeping track of what changes to variables are made by each statement. In the table below, statements are referred to by the numbers labeling the lines in the code above. We can track the state of each variable after each line in executed. A dash is shown where a variable is not defined. For instance after line 1 is executed, a value is given to x, but y is still undefined. Then y gets a value in line 2. The comment on the right summarizes what is happening. Since x has the value 3 when line 2 starts, x+2 is the same as 3+2. In line three we use the fact that the right side of an assignment statement uses the values of variables when the line starts executing (what is left after the previous line of the table executed), but the assignment to the variable y on the left causes a change to y, and hence the updated value of y, 10, is shown in the table. Line 4 then changes x, using the *latest* value of y (10, not the initial value 5!). The result from line 5 confirms the values of x and y.

Line	x	y	comment
1	3	-	
2	3	5	5=3+2, using the value of x from the previous line
3	3	10	10=2*5 on the right, use the value of y from the previous line
4	7	19	7=10-3 on the right, use the value of x and y from the previous line
5	7	10	print: 7 10

The order of execution will always be the order of the lines in the table. In this simple sequential code, that also follows the textual order of the program. Following each line of execution of a program in order, carefully, keeping track of the current values of variables, will be called *playing computer*. A table like the one above is an organized way to keep track.

[6]It is possible with function recursion, but we will avoid that topic in this introduction.

1.13.2. The `list` Type. Lists are ordered sequences of arbitrary data. Lists are the first kind of data discussed so far that are *mutable*: the length of the sequence can be changed and elements substituted. We will delay the discussion of changes to lists until a further introduction to objects. Lists can be written explicitly. *Read* the following examples

```
['red', 'green', 'blue']
[1, 3, 5, 7, 9, 11]
['silly', 57, 'mixed', -23, 'example']
[] # the empty list
```

The basic format is square-bracket-enclosed, comma-separated lists of arbitrary data.

1.13.3. The `range` Function, Part 1. There is a built-in function `range`, that can be used to automatically generate regular arithmetic sequences. Try the following in the *Shell*:

```
list(range(4))
list(range(10))
```

The general pattern for use is

```
range(sizeOfSequence)
```

This syntax will generate the items, one at a time, as needed. If you want to see all the results at once as a list, you can convert to a list as in the examples above. The resulting sequence starts at 0 and ends *before* the parameter. We will see there are good reasons to start from 0 in Python. One important property of sequences generated by `range(n)` is that the total number of elements is `n`. The sequence omits the number `n` itself, but includes 0 instead.

With more parameters, the `range` function can be used to generate a much wider variety of sequences. The elaborations are discussed in Section 2.4.12 and Section 3.3.2.

1.13.4. Basic `for` Loops. Try the following in the *Shell*. You get a sequence of continuation lines before the Shell responds. Be sure to indent the second and third lines. (This is only needed inthe Shell, not in an edit window, where the indentation is automatic). *Be sure to enter another empty line (just* ENTER*) at the end to get the Shell to respond.*

```
for count in [1, 2, 3]:
    print(count)
    print('Yes' * count)
```

This is a `for` loop. It has the heading starting with `for`, followed by a variable name (`count` in this case), the word `in`, some sequence, and a final colon. As with function definitions and other heading lines ending with a colon, the colon at the end of the line indicates that a consistently indented block of statements follows to complete the `for` loop.

```
for item in sequence:
    indented statements to repeat
```

The block of lines is repeated once for each element of the sequence, so in this example the two lines in the indented block are repeated three times. Furthermore the variable in the heading (`count` here) may be used in the block, and each time through it takes on the *next* value in the sequence, so the first time through the loop `count` is 1, then 2, and finally 3. Look again at the output and see that it matches this sequence.

There is a reason the interpreter waited to respond until after you entered an empty line: The interpreter did not know how long the loop block was going to be! The empty line is a signal to the interpreter that you are done with the loop block.

Look at the following example program `for123.py`, and run it.

```
for count in [1, 2, 3]:                #1
    print(count)                       #2
    print('Yes'*count)                 #3
print('Done counting.')                #4
for color in ['red', 'blue', 'green']:  #5
    print(color)                       #6
```

In a file, where the interpreter does not need to respond immediately, the blank line is not necessary. Instead, as with a function definition or any other format with an indented block, you indicate being past the indented block by *de*denting to line up with the `for`-loop heading. Hence in the code above, "Done Counting." is printed once after the first loop completes all its repetitions. Execution ends with another simple loop.

As with the indented block in a function, it is important to get the indentation right. Alter the code above, so line 4 is indented:

```
for count in [1, 2, 3]:                 #1
    print(count)                        #2
    print('Yes'*count)                  #3
    print('Done counting.')                 #4
for color in ['red', 'blue', 'green']:  #5
    print(color)                        #6
```

Predict the change, and run the code again to test.

Loops are one of the most important features in programming. While the syntax is pretty simple, using them creatively to solve problems (rather than just look at a demonstration) is among the biggest challenges for many learners at an introductory level. One way to simplify the learning curve is to classify common situations and patterns. One of the simplest patterns is illustrated above, simple *for-each* loops.

for *item* in *sequence*
    do some thing with *item*

(It would be even more like English if `for` were replace by `for each`, but the shorter version is the one used by Python.)

In the `for`-loop examples above, something is printed that is related to each item in the list. Printing is certainly one form of "do something", but the possibilities for "do something" are completely general!

We can use a for-each loop to revise our first example. *Recall* the code from madlib.py:

```
addPick('animal', userPicks)
addPick('food', userPicks)
addPick('city', userPicks)
```

Each line is doing exactly the same thing, except varying the string used as the cue, while repeating the rest of the line. This is the for-each pattern, but we need to list the sequence that the cues come from. *Read* the alternative:

```
for cue in ['animal', 'food', 'city']:  # heading
    addPick(cue, userPicks)             # body
```

If you wish to see or run the whole program with this small modification, see the example `madlibloop.py`.

It is important to understand the sequence of operations, how execution goes back and forth between the heading and the body. Here are the details:

(1) heading first time: variable `cue` is set to the first element of the sequence, `'animal'`
(2) body first time: since `cue` is now `'animal'`, effectively execute `addPick('animal', userPicks)` (Skip the details of the function call in this outline.)
(3) heading second time: variable `cue` is set to the next element of the sequence, `'food'`
(4) body second time: since `cue` is now `'food'`, effectively execute `addPick('food', userPicks)`
(5) heading third time: variable `cue` is set to the next (last) element of the sequence, `'city'`
(6) body third time: since `cue` is now `'city'`, effectively execute `addPick('city', userPicks)`
(7) heading done: Since there are no more elements in the sequence, the entire `for` loop is done and execution would continue with the statement after it.

This looping construction would be even handier if you were to modify the original mad lib example, and had a story with many more cues. Also this revision will allow for further improvements in Section 2.3.3, after we introduce more about string manipulation.

1.13.5. Simple Repeat Loops. The examples above all used the value of the variable in the `for`-loop heading. An even simpler `for`-loop usage is when you just want to repeat the exact same thing a specific number of times. In that case only the *length* of the sequence, not the individual elements are important. We have already seen that the `range` function provides an ease way to produce a sequence with a specified number of elements. Read and run the example program `repeat1.py`:

```
for i in range(10):
    print('Hello')
```

In this situation, the variable `i` is not used inside the body of the for-loop.

The user could choose the number of times to repeat. Read and run the example program `repeat2.py`:

```
n = int(input('Enter the number of times to repeat: '))
for i in range(n):
    print('This is repetitious!')
```

1.13.6. Successive Modification Loops. Suppose I have a list of items called `items`, and I want to print out each item and number them successively. For instance if `items` is ['red', 'orange', 'yellow', 'green'], I would *like* to see the *output*:

```
1 red
2 orange
3 yellow
4 green
```

Read about the following thought process for developing this:

If I allow myself to omit the numbers, it is easy: For any `item` in the list, I can process it with

```
print(item)
```

and I just go through the list and do it *for each* one. (Copy and run if you like.)

```
items = ['red', 'orange', 'yellow', 'green']
for item in items:
    print(item)
```

Clearly the more elaborate version with numbers has a pattern with some consistency, each line is at least in the form:

```
number item
```

but the number changes each time, and the numbers do *not* come straight from the list of items.

A variable can change, so it makes sense to have a variable `number`, so we have the potential to make it change correctly. We could easily get it right the first time, and then repeat the *same* number. Read and run the example program `numberEntries1.py`:

```
items = ['red', 'orange', 'yellow', 'green']
number = 1
for item in items:
    print(number, item)
```

Of course this is still not completely correct, since the idea was to *count*. After the first time number is printed, it needs to be changed to 2, to be right the next time through the loop, as in the following code: Read and run the example program `numberEntries2.py`:

```
items = ['red', 'orange', 'yellow', 'green']
number = 1
for item in items:
    print(number, item)
    number = 2
```

This is closer, but still not completely correct, since we never get to 3! We need a way to change the value of number *that will work each time through the loop*. The pattern of counting is simple, so simple in fact that you probably do not think consciously about how you go from one number to the next: You can describe the pattern by saying each successive number is *one more than the previous number*. We need to be able to change `number` so it is one more than it was before. That is the additional idea we need! Change the last line of the loop body to get the example program numberEntries3.py. See the addition and run it:

```
items = ['red', 'orange', 'yellow', 'green']  #1
number = 1                                    #2
for item in items:                            #3
    print(number, item)                       #4
    number = number + 1                       #5
```

It is important to understand the step-by-step changes during execution. Below is another table showing the results of playing computer. The line numbers are much more important here to keep track of the flow of control, because of the jumping around at the end of the loop.

Line	items	item	number	comment
1	['red', 'orange', 'yellow', 'green']	-	-	
2	['red', 'orange', 'yellow', 'green']	-	1	
3	['red', 'orange', 'yellow', 'green']	'red'	1	start with item as first in sequence
4	['red', 'orange', 'yellow', 'green']	'red'	1	print: 1 red
5	['red', 'orange', 'yellow', 'green']	'red'	2	2 = 1+1
3	['red', 'orange', 'yellow', 'green']	'orange'	2	on to the next element in sequence
4	['red', 'orange', 'yellow', 'green']	'orange'	2	print 2 orange
5	['red', 'orange', 'yellow', 'green']	'orange'	3	3=2+1
3	['red', 'orange', 'yellow', 'green']	'yellow'	3	on to the next element in sequence
4	['red', 'orange', 'yellow', 'green']	'yellow'	3	print 3 yellow
5	['red', 'orange', 'yellow', 'green']	'yellow'	4	4=3+1
3	['red', 'orange', 'yellow', 'green']	'green'	4	on to the last element in sequence
4	['red', 'orange', 'yellow', 'green']	'green'	4	print 4 green
5	['red', 'orange', 'yellow', 'green']	'green'	5	5=4+1
3	['red', 'orange', 'yellow', 'green']	'green'	5	sequence done, end loop and code

The final value of number is never used, but that is OK. What we want is printed.

This short example illustrates a lot of ideas:

- Loops may contain several variables.
- One way a variable can change is by being the variable in a for-loop heading, that automatically goes through the values in the for-loop list.
- Another way to have variables change in a loop is to have an explicit statement that changes the variable inside the loop, causing *successive modifications*.

There is a general pattern to loops with successive modification of a variable like `number` above:

(1) The variables to be modified need *initial* values *before* the loop (line 1 in the example above).
(2) The loop heading causes the repetition. In a for-loop, the number of repetitions is the same as the size of the list.
(3) The body of the loop generally "does something" (like print above in line 4) that you want done repeatedly.
(4) There is code inside the body of the loop to set up for the *next* time through the loop, where the variable which needs to change gets transformed to its next value (line 5 in the example above).

This information can be put in a code outline:

```
Initialize variables to be modified
Loop heading controlling the repetition
    Do the desired action with the current variables
    Modify variables to be ready for the action the next time
```

If you compare this pattern to the for-each and simple repeat loops in Section 1.13.4, you see that the examples there were simpler. There was no explicit variable modification needed to prepare for the next time though the loop. We will refer to the latest, more general pattern as a *successive modification* loop.

Functions are handy for encapsulating an idea for use and reuse in a program, and also for testing. We can write a function to number a list, and easily test it with different data. Read and run the example program `numberEntries4.py`:

```
def numberList(items):
    '''Print each item in a list items, numbered in order.'''
    number = 1
    for item in items:
        print(number, item)
        number = number + 1
```

```
def main():
    numberList(['red', 'orange', 'yellow', 'green'])
    print()
    numberList(['apples', 'pears', 'bananas'])


main()
```

Make sure you can follow the whole sequence, step by step! This program has the most complicated flow of control so far, changing both for function calls and loops.

(1) Execution start with the very last line, since the previous lines are definitions
(2) Then `main` starts executing.
(3) The first call to `numberList` effectively sets the formal parameter
`items = ['red', 'orange', 'yellow', 'green']`
and the function executes just like the flow followed in `numberEntries3.py`. This time, however, execution returns to main.
(4) An empty line is printed in the second line of main.
(5) The second call to `numberList` has a different actual parameter `['apples', 'pears', 'bananas']`, so this effectively sets the formal parameter this time
`items = ['apples', 'pears', 'bananas']`
and the function executes in a similar pattern as in `numberEntries3.py`, but with different data and one less time through the loop.
(6) Execution returns to main, but there is nothing more to do.

1.13.7. Accumulation Loops. Suppose you want to add up all the numbers in a list, `nums`. Let us plan this as a function from the beginning, so *read* the code below. We can start with:

```
def sumList(nums):
    '''Return the sum of the numbers in nums.'''
```

If you do not see what to do right away, a useful thing to do is write down a *concrete* case, and think how you would solve it, in complete detail. If `nums` is `[2, 6, 3, 8]`, you would likely calculate

2+6 is 8
8 + 3 is 11
11 + 8 is 19
19 is the answer to be returned.

Since the list may be arbitrarily long, you need a loop. Hence you must find a pattern so that you can keep reusing the *same statements* in the loop. Obviously you are using each number in the sequence in order. You also generate a sum in each step, which you reuse in the next step. The pattern is different, however, in the first line, 2+6 is 8: there is no previous sum, and you use two elements from the list. The 2 is not added to a previous sum.

Although it is not the shortest way to do the calculation *by hand*, 2 is a sum of 0 + 2: We can make the pattern consistent and calculate:

start with a sum of 0
0 + 2 is 2
2 + 6 is 8
8 + 3 is 11
11 + 8 is 19
19 is the answer.

Then the second part of each sum is a number from the list, `nums`. If we call the number from the list `num`, the main calculation line in the loop could be

```
nextSum = sum + num
```

The trick is to use the same line of code the next time through the loop. That means what was `nextSum` in one pass becomes the `sum` in the next pass. One way to handle that is:

```
sum = 0
for num in nums:
    nextSum = sum + num
    sum = nextSum
```

Do you see the pattern? Again it is

```
initialization
loop heading
    main work to be repeated
    preparation for the next time through the loop
```

Sometimes the two general loop steps can be combined. This is such a case. Since `nextSum` is only used once, we can just substitute its value (`sum`) where it is used and simplify to:

```
sum = 0
for num in nums:
    sum = sum + num
```

so the whole function, with the `return` statement is:

```
def sumList(nums):                                  #1
    '''Return the sum of the numbers in nums.'''
    sum = 0                                         #2
    for num in nums:                                #3
        sum = sum + num                             #4
    return sum                                      #5
```

With the following (not indented) line below used to test the function, you have the example program sumNums.py. Run it.

```
print(sumList([5, 2, 4, 7]))
```

The pattern used here is certainly successive modification (of the `sum` variable). It is useful to give a more specialized name for this version of the pattern here. It follows an *accumulation* pattern:

```
initialize the accumulation to include none of the sequence (sum = 0 here)
for item in sequence :
    new value of accumulation = result of combining item with last value of accumulation
```

This pattern will work in many other situations besides adding numbers.

Exercise 1.13.7.1. * Suppose the function `sumList`, is called with the parameter [5, 2, 4, 7]. Play computer on this call. Make sure there is a row in the table for each line executed in the program, each time it is executed. In each row enter which program line is being executed and show all changes caused to variables by the execution of the line. A table is started for you below. The final line of your table should be for line 5, with the comment, "return 18". If you do something like this longhand, and the same long value repeats a number of times, it is more convenient to put a ditto (") for each repeated variable value or even leave it blank. If you want to do it on a computer you can start from the first table in example file `playComputerSumStub.rtf`. First save the file as playComputerSum.rtf.

Line	nums	sum	num	comment
1	[5, 2, 4, 7]	-	-	
2				

EXERCISE 1.13.7.2. * Write a program `testSumList.py` which includes a `main` function to test the sumList function several times. Include a test for the extreme case, with an empty list.

EXERCISE 1.13.7.3. ** Complete the following function. This starting code is in `joinAllStub.py`. Save it to the *new* name `joinAll.py`. Note the way an example is given in the documentation string. It simulates the use of the function in the Shell. This is a common convention:

```
def joinStrings(stringList):
    '''Join all the strings in stringList into one string,
    and return the result. For example:

    >>> print(joinStrings(['very', 'hot', 'day']))
    'veryhotday'
    '''
```

Hint1: [7] Hint2: [8]

1.13.8. More Playing Computer. Testing code by running it is fine, but looking at the results does not mean you really understand what is going on, particularly if there is an error! People who do not understand what is happening are likely to make random changes to their code in an attempt to fix errors. This is a *very bad*, increasingly self-defeating practice, since you are likely to never learn where the real problem lies, and the same problem is likely to come back to bite you.

It is important to be able to predict accurately what code will do. We have illustrated playing computer on a variety of small chunks of code.

Playing computer can help you find bugs (errors in your code). Some errors are syntax errors caught by the interpreter in translation. Some errors are only caught by the interpreter during execution, like failing to have a value for a variable you use. Other errors are not caught by the interpreter at all – you just get the wrong answer. These are called *logical* errors. Earlier logical errors can also trigger an execution error later. This is when playing computer is particularly useful.

A common error in trying to write the `numberList` function would be to have:

```
def numberList(items):    # WRONG code for illustration!!!!    #1
    '''Print each item in a list items, numbered in order.'''  #2
    for item in items:                                         #3
        number = 1                                             #4
        print(number, item)                                    #5
        number = number + 1                                    #6
```

You can run this code in `numberEntriesWRONG.py` and see that it produces the wrong answer. If you play computer on the call to `numberList(['apples', 'pears', 'bananas'])`, you can see the problem:

[7] This is a form of accumulation, but not quite the same as adding numbers.

[8] "Start with nothing accumulated" does not mean 0, here. Think what is appropriate.

Line	items	item	number	comment
1	['apples', 'pears', 'bananas']	-	-	pass actual parameter value to items
3	['apples', 'pears', 'bananas']	'apples'	-	start with item as first in sequence
4	['apples', 'pears', 'bananas']	'apples'	1	
5	['apples', 'pears', 'bananas']	'apples'	1	print: 1 apples
6	['apples', 'pears', 'bananas']	'apples'	2	2 = 1+1
3	['apples', 'pears', 'bananas']	'pears'	2	on to the next element in sequence
4	['apples', 'pears', 'bananas']	'apples'	1	
5	['apples', 'pears', 'bananas']	'pears'	1	print: 1 pears *OOPS!*

If you go step by step you should see where the incorrect 1 came from: the initialization is repeated each time in the loop at line 4, undoing the incrementing of `number` in line 6, messing up your count. Always be careful that your *one-time* initialization for a loop goes *before* the loop, not in it!

Functions can also return values. Consider the Python for this mathematical sequence: define the function m(x) = 5x, let y = 3; find m(y) + m(2y-1).

```
def m(x):                   #1
    return 5*x              #2

y = 3                       #3
print(m(y) + m(2*y-1))      #4
```

A similar example was considered in Section 1.11.6, but now add the idea of playing computer and recording the sequence in a table. Like when you simplify a mathematical expression, Python must complete the innermost parts first. Tracking the changes means following the function calls carefully and using the values returned. Again a dash '-' is used in the table to indicate an undefined variable. Not only are local variables like formal parameters undefined before they are first used, they are also undefined after the termination of the function,

Line	x	y	comment
3	-	3	(definitions only before this line)
4	-	3	start on: print m(y) + m(2*y-1); find m(y), which is m(3)
1	3	3	pass 3 to function m, so x =3
2	3	3	return 5*3 = 15
4	-	3	substitute result: print 15 + m(2*y-1), find m(2*y-1), which is m(2*3-1) = m(5)
1	5	3	pass 5 to function m, so x=5
2	5	3	return 5*5 = 25
4	-	3	substitute result: print 15 + 25, so calculate and print 40

Thus far most of the code given has been motivated first, so you are likely to have an idea what to expect. You may need to read code written by someone else (or even yourself a while back!) where you are not sure what is intended. Also you might make a mistake and accidental write code that does something unintended! If you really understand how Python works, one line at a time, you should be able to play computer and follow at least short code sequences that have not been explained before. It is useful to read another person's code and try to follow it. The next exercises also provides code that has not been explained first:or has a mistake.

Exercise 1.13.8.1. ** Play computer on the following code. Reality check: 31 is printed when line 6 finally executes. Table headings are shown below to get you started with a pencil. Alternately you can work in a word processor starting from playComputerStub.rtf, which has tables set up for this and the following exercise. Save the file with an alternate name playComputer.rtf.

```
x = 0                   #1
y = 1                   #2
for n in [5, 4, 6]:     #3
    x = x + y*n         #4
    y = y + 1           #5

print(x)                #6
```

Line	x	y	n	Comment

EXERCISE 1.13.8.2. ** The following code is supposed to compute the product of the numbers in a list. For instance `product([5, 4, 6])` should calculate and return 5*4*6=120 in steps, calculating 5, 5*4=20 and 20*6=120 . Play computer on a call to `product([5, 4, 6])` until you see that it makes a mistake. This code appears in the example file `numProductWrong.py`. Save it as `numProduct.py` and fix the error (and save again!). Table headings and the first row are shown below to get you started with a pencil. Alternately you can work in a word processor continuing to add to playComputer.rtf, started in the previous exercise.

```
def product(nums):      #1
    for n in nums:      #2
        prod = 1        #3
        prod = prod*n   #4
    return prod         #5
```

Line	nums	n	prod	Comment
1	[5, 4, 6]	-	-	

EXERCISE 1.13.8.3. ** Play computer on the following code. Table headings are shown for you. Reality check: 70 is printed. See the previous exercises if you enter your answer in a file.

```
def f(x):                  #1
  return x+4               #2

print(f(3)*f(6))           #3
```

Line	x	Comment

1.13.9. The `print` function `end` keyword. By default the print function adds a newline to the end of the string being printed. this can be overridden by including the keyword parameter `end`. The keyword end can be set equal to any string. The most common replacements are the empty string or a single blank. If you also use the keyword parameter `sep`, these keyword paramters may be in either order, but they msut come at the end of the parmater list. Read the illustrations:

```
print('all', 'on', 'same', 'line')
print('different line')
```

is equivalent to

```
print('all', 'on' , end = ' ')
print('same', end = ' ')
print('line')
print('different line')
```

This does not work directly in the shell (where you are always forced to a new line at the end). It does work in a program, but it is not very useful except in a loop! Suppose I want to print a line with all the elements of a list, separated by spaces, but not on separate lines. I can use the `end` keyword set to a space in the loop. Can you figure out in your head what this example file `endSpace1.py` does? Then try it:

```
def listOnOneLine(items):
    for item in items:
         print(item, end=' ')

listOnOneLine(['apple', 'banana', 'pear'])
print('This may not be what you expected!')
```

If you still want to go on to a new line at the *end* of the loop, you must include a print function that does advance to the next line, once, *after* the loop. Try this variation, `endSpace2.py`

```
def listOnOneLine(items):
    for item in items:
        print(item, end=' ')
    print()

listOnOneLine(['apple', 'banana', 'pear'])
print('This is probably better!')
```

1.14. Decimals, Floats, and Floating Point Arithmetic

Floating point numbers like 12.345 are a basic type, but there are some complications due to their inexactness. This section may be deferred until you actually need numbers other than integers.

1.14.1. Floats, Division, Mixed Types. As you moved on in school after your first integer division, and did fractions and decimals, you probably thought of 6/8 as a fraction and could convert to a decimal .75. Python can do decimal calculations, too, *approximately.*

Try all set-off lines in this section in the Shell:

```
6/8
6/3
2.3/25.7
```

There is more going on here than meets the eye. As you should know, decimal representations of values can be a pain. They may not be able to be expressed with a finite number of characters. Try

```
2/3
```

Also, as you may have had emphasized in science class, real number measurements are often not exact, and so the results of calculations with them are not exact. In fact there are an infinite number of real number just between 0 and 1, and a computer is finite. It cannot store all those numbers exactly! On the other hand, Python does store *integers* exactly (well at least far past the number of atoms in the universe – eventually even integers could get too big to store in a computer). The difference in the way integers and decimals are stored and processed leads to decimals and integers being different *type*s in Python. Try

```
type(3.5)
```

Note that 3.5 is of type 'float', not 'decimal'. There are several reasons for that name having to do with the actual way the type is stored internally. "Decimal" implies base ten, our normal way for writing numbers with ten digits 0,1,2,3,4,5,6,7,8,9. Computers actually use base two, with only two symbols 0,1. (Did you note what symbols were in the machine language in Section 1.1?) Also floats use an encoding something like scientific notation from science class, with exponents that allow the decimal point to move or "float", as in the decimal case: $2345.6 = (2.3456)10^3$

Try

```
type(-2)
type(-2.0)
```

Even a number that is actually an integer can be represented in the float type if a decimal point is included.

Always be sure to remember that floats may not be exact. The use of base two makes this true even in cases where decimal numbers *can* be expressed exactly! More on that at the end of this section on formatting floats.

It is sometimes important to know the numeric type of a Python value. Any combination of +, -, and * with operands of type int produces an int. If there is an operation /, or if any operand is of type float, the result is float. Try each in the *Shell* (and guess the resulting type):[9]

```
3.3 - 1.1
2.0 + 3
2.5*2
```

[9]Python 3.1 does what you would expect mathematically with an expression like

```
(1/2)*6.5
```

Caution: This is not the case in other common languages like Java and C++ (or with versions of Python before 3.0). They treat the / operation with integers like the current Python //, so the result of the expression above is 0!

1.14.2. Exponentiation, Square Roots. Exponentiation is finding powers. In mathematical notation, $(3)(3)(3)(3) = 3^4 = 81$. In Python there is no fancy typography with raised exponent symbols like the 4, so Python uses ** before a power: Try in the *Shell*:

```
3**4
5*2**3
```

If you expected 1000 for the second one, remember exponentiation has even higher precedence than multiplication and division: 2**3 is 2*2*2 or 8, and 5*8 is 40.

Exponents do not need to be integers. A useful example is the 0.5 power: it produces a square root. Try in the *Shell*:

```
9**.5
2**.5
```

The result of a power operation is of int type only if both parameters are integers and the correct result is an integer.

1.14.3. String Formats for Float Precision. You generally do not want to display a floating point result of a calculation in its raw form, often with an enormous number of digits after the decimal point, like 23.457413902458498. You are likely to prefer rounding it to something like 23.46. There are two approaches.

First there is a format *function* (not method) with a second parameter allowed to specialize the formatting of objects as strings.. *Read* the following example interpreter sequence showing possibilites when a float is being formatted:

```
>>> x = 23.457413902458498
>>> format(x, '.5f')
>>> '23.45741'
>>> format(x, '.2f')
>>> '23.46'
```

Note that the results are rounded not *truncated*: the result to two places is 23.46, not 23.45. The formatting string `'.5f'` means after the decimal point round to 5 places. Similarly `'.2f'` means round to two decimal places.

This rounding notation can also be placed after a colon inside the braces of format strings, for use with the string format *method*. Read the Shell session:

```
>>> x = 2.876543
>>> 'longer: {x:.5f}, shorter: {x:.3f}.'.format(**locals())
>>> 'longer: 2.87654, shorter: 2.877.'
```

The colon separates the symbol identifying what value to use for the substitution from the instructions for the specific formating method.

The colon and formatting instructions can also be used with the format versions depending on the order of the parameters. Continuing the earlier *Shell* example:

```
>>> 'No dictionary: {:.5f}.'.format(x)
>>> 'No dictionary: 2.87654.'
```

There are many more fancy formatting options for the string `format` method that we will not discuss.

Going to the opposite extreme, and using formatting with many digits, you can check that Python does not necessarily remember simple decimal numbers exactly:

```
>>> format(.1, '.20f')
'0.10000000000000000555'
>>> format(.2, '.20f')
'0.20000000000000001110'
>>> format(.1 + .2, '.20f')
'0.30000000000000004441'
>>> format(.3, '.20f')
'0.29999999999999998890'
>>>
```

Python stores the numbers correctly to about 16 or 17 digits. You may not care about such slight errors, but you will be able to check in Chapter 3 that if Python tests the expressions .1 + .2 and .3 for equality, it decides that they are not equal! In fact, as you can see above, the approximations that Python stores for the two expressions are *not* exactly equal. Do not depend on the exactness of floating point arithmetic, even for apparently simple expressions!

The floating point formatting code in this section is also in example program `floatFormat.py`.

EXERCISE 1.14.3.1. * Write a program, `discount.py`, that prompts the user for an original price and for a discount percentage and prints out the new price to the nearest cent. For example if the user enters 2.89 for the price and 20 for the discount percentage, the value would be (1- 20/100)*2.89, rounded to two decimal places, 2.31. For price .65 with a 25 percent discount, the value would be (1- 25/100)*.65, rounded to two decimal places, .49.[10] Write the general calculation code following the pattern of the calculations illustrated in the two concrete examples.

1.15. Summary

Section numbers in square brackets indicate where an idea was first discussed.

Where Python syntax is illustrated, the typeface indicates the the category of each part:

Typeface	Meaning
`Typewriter font`	Text to be written verbatim
Emphasized	A place where you can use an arbitrary identifier. The emphasized text attempts to be descriptive of the meaning of the identifier in the current context.
Normal text	A description of what goes in that position, without giving explicit syntax

If there are several variations on a particular part of the syntax, alternatives will be show on successive lines.

To emphasize the successive parts of the syntax, space will generally be left around symbol and punctuation characters, but the space is not required in actual use.

(1) Python Shell
  (a) A Shell window may be opened from the Idle menu: Run -> Python Shell [1.2.5]
  (b) Entering commands:
    (i) Commands may be entered at the `>>>` prompt. [1.4.1]
    (ii) If the Shell detects that a command is not finished at the end of the line, a continuation line is shown with no `>>>`. [1.4.2]
    (iii) Statements with a heading ending in a colon followed by an indented block, must be terminated with an empty line. [1.13.4]
    (iv) The Shell evaluates a completed command immediately, displaying any result other than `None`, starting on the next line. [1.4.1]
    (v) The Shell remembers variable and function names. [1.6]
  (c) An earlier Shell line may to copied and edited by clicking anywhere in the previously displayed line and then pressing ENTER.
(2) Idle editing
  (a) Start a new window from the File menu by selecting New, Open..., or Recent Files. [1.9.1]
  (b) Make your Python file names explicitly end with '.py' [1.6.1]
(3) To run a program from an Idle Editor Window:
  (a) Select Run -> Run Module or press function key F5. The program runs in the Shell window, after resetting the shell so all old names are forgotten. [1.9.1]
    (i) If the program is expecting keyboard input, the text cursor should appear at the end of the Shell history. If you somehow move the cursor elsewhere, you must explicitly move it back. [1.9]

[10]In Python 3.0+, the previous expressions make sense, but in earlier versions of Python and in other languages like C++ and Java, where there are not separate division operators // and /, these expressions would be wrong because of the multiple meanings of the operator / with different types. The expressions would work in these other languages if, for example, 100 were replaced by 100.0.

(ii) BUG WORKAROUND: If you were running a program that was expecting keyboard input when you terminated it to start the latest run, you will need to start by pressing the Enter key once or maybe twice to clear the old pending wait for input. [1.9.2]
(iii) Press Ctrl-C to stop a running program in a long or infinite loop.
(iv) After a program terminates, the Shell remembers function definitions and variable names define outside of any function. [1.11.2]

(4) Errors come in three categories:
 (a) Syntax errors: text that the interpreter recognizes as illegal when first reading it. This prevents execution of your code. Python lets you know where it *realized* there was an error. Sometimes this is the exact location, but the actual error could be anywhere earlier, often on the previous line. [1.6]
 (b) Execution errors: The first illegal action is detected while running your command or program. The source of the error could be in the line where execution fails, or it could be an earlier logical error that only later forces an execution error. [1.6]
 (c) Logical errors: When Python detects nothing illegal, but you do not get the results you desire. These errors are the hardest to trace down. Playing computer and additional print functions help. [1.13.8]

(5) Type `int`, (short for integer):
 (a) Literal integer values may not contain a decimal point. [1.14.1]
 (b) Integers may be arbitrarily large and are stored exactly. [1.14.1]
 (c) Integers have normal operations, with usual precedence (highest listed first):
  (i) `**`: exponentiation (5**3 means 5*5*5) [1.14.2]
  (ii) `*`, `/`,`//`, `%`: multiplication, division with float result, integer division (ignoring any remainder), just the remainder from division [1.4.3]
  (iii) `+`, `-`: addition, subtraction [1.4.1]

(6) Type `float`, (short for floating point): approximations of real numbers
 (a) Literal values must contain a decimal point to distinguish them from the int type [1.14.1]
 (b) Approximates a wide range of values [1.14.1]
 (c) Does not dependably store numbers exactly – even numbers with simple decimal representation [1.14.1]
 (d) Has the same operation symbols as for integers [1.14.1]
 (e) A mixed binary operation with an integer and a `float` produces a `float` result. [1.14.1]

(7) Type `str`, (short for string):
Literal values contain a sequence of characters enclosed in matching quotes.
 (a) Enclosed in `'`or `"`: The string must be on one line. [1.5.1]
 (b) Enclosed in `'''`or `"""`: The string may include multiple lines in the source file. [1.8.1]
 (c) Escape codes inside literals include `\'` for a single quote and `\n` for a newline. [1.8.2]
 (d) Binary operations (operation symbols have the same precedence order as when the symbols are used in arithmetic)
  (i) stringExpression1 + stringExpression2
  concatenation (running together) of the two strings [1.5.2]
  (ii) stringExpression `*` integerExpression
  integerExpression `*` stringExpression
  Repeat the string the number of times given by the integer expression. [1.5.2]
 (e) string format method:
  (i) `*stringFormatExpression*.format(*parameter0, parameter1, parameter2, ...*)` [1.10.4]
  where stringFormatExpression is any string with an arbitrary number of formatted substitutions in it. Formatted substitutions are enclosed in braces. A digit inside the braces will indicate which parameter value is substituted, counting from 0. If digits are left out, the format paramters are substituted in order. The expression inside the braces can end with a colon (:) followed by a format specifying string such as:
  `.#f` where # can be a non negative integer: substitute a numerical value rounded to the specified number of places beyond the decimal point. [1.14.1]

Example: `'A word: {}, a number: {}, a formatted number: {:.3f}.'.format('Joe', 23, 2.13579)`
evaluates to: `'A word: Joe, a number: 23, a formatted number: 2.136.'`

(ii) *`stringFormatExpression`*`.format(`*`**dictionary)`* The format expressions are the same as above except that a key name from a dictionary appears inside the braces instead of a digit. The dictionary referenced appears int he parameter list preceded by `**`. The value to be substituted is then taken from the dictionary by accessing the key. Example: If `defs` is a dictionary with `defs['name']` equaling `'Joe'`, `defs['num']` equaling 23, `defs['dec']` equaling 2.13579, then
`'A word: {name}, a number: {num}, a formatted number: {dec:.3f}.'.format(**defs)`
evaluates to the same string as in the previous example. [1.12.2]

(f) Strings are a kind of sequence.

(8) Type `list`
`[` expression `,` expression `,` and so on `]`
`[` expression `]`
`[]`
(a) A literal list consists of a comma separated collection of values all enclosed in square brackets. There may be many, one, or no elements in the list. [1.13.2]
(b) A list is a kind of sequence, so it may be used as the sequence in a for-statement heading. [1.13.4]

(9) Type `dict` (short for dictionary)
`dict()`
returns an empty dictionary
(a) A dictionary provides an association of each key to its value. The key can be any immutable type, with includes numbers and strings. [1.12.1]
(b) *dictName* [keyExpression] = valueExpression
associates in the dictionary *dictName* the key derived from evaluating keyExpression with the value derived from evaluating valueExpression. [1.12.1]
(c) Used in an expression, *dictName* [keyExpression] evaluates to the value in the dictionary *dictName* coming from the key obtained by evaluating keyExpression. [1.12.1]

(10) Type of `None`: This literal value has its own special type. `None` indicates the absence of a regular object.

(11) Identifiers
(a) Identifiers are names for Python objects [1.6.1]
(b) They may only contain letters, digits, and the underscore, and cannot start with a digit. They are case sensitive. [1.6.1]
(c) You cannot use a reserved word as an identifier, nor are you recommended to redefine an identifier predefined by Python. In the Idle editor you are safe if your identifier names remain colored black. [1.6.1]
(d) By convention, multi-word identifiers either [1.6.1]
(i) use underscores in place of blanks (since blanks are illegal is identifiers), as in `initial_account_balance`
(ii) use camelcase: all lowercase except for the starting letter of the second and later words, as in `initialAccountBalance`

(12) Variables are identifiers used to name Python data [1.6]
(a) When a variable is used in an expression, its latest value is substituted. [1.6]

(13) Statements
(a) Assignment statement: [1.6]
variable `=` expression
(i) The expression on the right is evaluated, using the latest values of all variables, and calculating all operations or functions specified.
(ii) The expression value is associated with the variable named on the left, removing any earlier association with the name.
(b) For-statement
```
for item in sequence :
```

consistently indented statement block, which may use the variable *`item`*

For each element in the sequence, repeat the statement block substituting the next element in the sequence for the name variable name *item*. See Programming Patterns for patterns of use. [1.13.4]

(c) Return statement
`return` expression
This is used only in a function definition, causing the function to immediately terminate and return the value of expression to the calling code, effectively acting as if the function call was replaced by this returned value. [1.11.6]

(14) Function calls
functionName `(` expression`,` expression`,` and so on `)`
  (a) The number of expressions must correspond to a number of parameters allowed by the function's definition. [1.11.4]
  (b) Even if there are no parameters, the parentheses must be included to distinguish the name of the function from a request to call the function. [1.11.2]
  (c) Each expression is evaluated and the values are passed to the code for the function, which executes its defined steps and may return a value. If the function call was a part of a larger expression, the returned value is used to evaluate the larger expression in the place where the function call was. [1.11.4]
  (d) If nothing is returned explicitly, the function returns `None`.
  (e) Function calls may also be used as statements, in which case any value that is returned is ignored (except if entered directly into the shell, which prints any returned value other than `None`).
  (f) Keyword arguments are a special case. They have been used optionally at the end of the parameter list for print.

(15) Functions that are built-in
  (a) Print function: [1.7] [1.13.9]
  `print(`expression)
  `print(`expression, expression, expression)
  `print(`expression, expression, expression, `sep=`stringVal, `end=`strVal)
  `print()`
    (i) Print the value of each expression in the list to the standard place for output (usually the screen) separating each value by individual blanks unless the keyword argument `sep` is specified to change it. There can be any number of expressions (not just 1 or 3 as illustrated)
    (ii) The string printed ends with a newline unless the keyword argument `end` is specified to change it.
    (iii) With no expression, the statement only advances to a new line.
  (b) Type names can be used as function to do obvious conversions to the type, as in `int('234')`, `float(123)`, `str(123)`. [1.10.3]
  (c) `type(`expression`)`
  Return the type of the value of the expression. [1.5.1]
  (d) `input`(promptString)
  Print the promptString to the screen; wait for the user to enter a line from the keyboard, ending with Enter. Return the character sequence as a string [1.10.1]
  (e) `len`(sequence)
  Return the number of elements in the sequence [1.3]
  (f) `range`(expression)
  Require expression to have a non negative integer value, call it n. Generate a sequence with length n, consisting of the numbers 0 through n-1. For example `range(4)` generates the sequence `0, 1, 2, and 3` [1.13.3]
  (g) `max`(expression1, expression2, and so on)
  Return the maximum of all the expressions listed. [1.3]

(h) `format`(expression, formatString) [1.14.1]
If expression is numeric, the format string can be in the form `'.#f'`, where the # gets replaced by a nonnegative integer, and the result is a string with the value of the expression rounded to the specified number of digits beyond the decimal point.

(16) Functions defined by a user:

`def` *`functionName`* `(` *`parameter1`*`,` *`parameter2`*`,` and so on`) :`
consistently indented statement block, which may include a return statement

(a) There may be any number of parameters. The parentheses must be included even if there are no parameters. [1.11.4]
(b) When a function is first defined, it is only remembered: its lines are not executed. [1.11.2]
(c) When the function is later called in other code, the actual parameters in the function call are used to initialize the local variables *parameter1*, *parameter2*, and so on in the same order as the actual parameters. [1.11.4]
(d) The local variables of a function are independent of the local names of any function defined outside of this function. The local variables must be initialized before use, and the names lose any association with their values when the function execution terminates. [1.11.8]
(e) If a return statement is reached, any further statements in the function are ignored. [1.11.6]
(f) Functions should be used to :
(i) Emphasize that the code corresponds to one idea and give an easily recognizable name. [1.11.2]
(ii) Avoid repetition. If a basic idea is repeated with just the data changing, it will be easier to follow and use if it is coded once as a function with parameters, that gets called with the appropriate actual parameters when needed. [1.11.4]
(iii) It is good to separate the internal processing of data from the input and output of data. This typically means placing the processing of data and the return of the result in a function. [1.11.4]
(iv) Separate responsibilities: The consumer of a function only needs to know the name, parameter usage, and meaning of any returned value. Only the writer of a function needs to know the implementation of a function. [1.11.7]

(17) Modules (program files)
(a) A module may start with a documentation string. [1.9.4]
(b) Define your functions in your module. If the module is intended as a main program called only one way, a convention is make your execution just be calling a function called main. [1.11.3]
(c) Avoid defining variable outside of your functions. Names for constant (unchanging) values are a reasonable exception. [1.11.9]

(18) Documentation String: A string, often a multi-line (triple quoted) string that may appear in two places:
(a) At the very beginning of a file: This should give overall introductory information about the file [1.9.4]
(b) As the very first entry in the body of a function: This should describe: [1.12.1]
(i) The return value of the function (if there is one)
(ii) Anything about the parameters that is not totally obvious from the names
(iii) Anything about the results from the function that is not obvious from the name

(19) Programming Patterns
(a) Input-calculate-Output: This is the simplest overall program model. First obtain all the data you need (for instance by prompting the user for keyboard input). Calculate what you need from this data. Output the data (for instance to the screen with print functions). [**??**]
(b) Repetitive patterns: These patterns are all associated with loops. Loops are essential if the number of repetitions depends on dynamic data in the program. Even if you could avoid a loop by repeating code, a loop is usually a better choice to make the repetitive logic of your program clear to all.
(i) Exact repetition some number of times: If the number of time to repeat is n:

`for` *`i`* `in range(`*`n`*`):`
actions to be repeated

Here the variable `i` is included only because there must be a variable name in a `for`-loop. [1.13.5]

(ii) For-each loop: Do the same sort of thing for each item in a specified sequence. [1.13.4]

```
for item in sequence :
    actions to be done with each item
```

(iii) Successive modification loop: Repeat a basic idea, but where the data involved each time changes via a pattern that is coded in the loop to convert the previous data into the data needed the next time through the loop [1.13.6]:

```
initialize all variables that will be successively modified in the loop
loop heading for the repetition :
    actions to be in each loop with the current variable values
    modify the variable values to prepare for the next time through the loop
```

(iv) Accumulation loop: A sequence of items need to be combined. This works where the accumulation of all the items can be approached incrementally, combining one after another with the accumulation so far [1.13.7]:

```
initialize the accumulation to include none of the sequence
for item in sequence :
    new value of accumulation =
            result of combining item with last value of accumulation
```

(20) Playing computer: testing that you understand your code (and it works right or helping you find where it goes wrong) [1.13.1,1.13.6, 1.13.8]
  (a) Make sure line numbers are labeled
  (b) Make a table with heading for line numbers, all variables that will be changing, and comments
  (c) Follow the order of execution, one statement at a time, being careful to update variable values and only use the latest variable values, and carefully following the flow of control through loops and into and out of function calls.

CHAPTER 2

Objects and Methods

2.1. Strings, Part III

2.1.1. Object Orientation. Python is an *object-oriented* language. Every piece of data and even functions and types are objects. The term object-oriented is used to distinguish Python from earlier languages, classified as *procedural* languages, where types of data and the operations on them were not connected in the language. The functions we have used so far follow the older procedural programming syntax. In the newer paradigm of object-oriented programming, all data are in objects, and a core group of operations that can be done on some particular type of object are tightly bound to the object and called the object's *methods*.

For example, strings are objects, and strings "know how" to produce an uppercase version of themselves. Try in the *Shell:*

```
s = 'Hello!'
s.upper()
```

Here `upper` is a *method* associated with strings. This means `upper` is a function that is bound to the string before the dot. This function is bound both logically, and as we see in the new notation, also syntactically. One way to think about it is that each type of data knows operations (methods) that can be applied to it. The expression `s.upper()` calls the method `upper` that is bound to the string `s` and returns a *new* uppercase string result based on `s`.

Strings are immutable, so no string method can change the original string, it can only return a new string. Confirm this by entering each line individually in the *Shell* to see the original s is unchanged:

```
s
s2 = s.upper()
s2
s
```

We are using the new object syntax:

object.*method*( )

meaning that the *method* associated with the object's type is applied to the *object*. This is just a special syntax for a function call with an object.

Another string method is `lower`, analogous to upper, but producing a lowercase result.

Test yourself: How would you write the expression to produce a lowercase version of the string *s*? Answer:[1] Try it in the ***Shell***.

Test yourself in the Shell: How would you use this string *s* and both the `lower` and `upper` methods to create the string `'hello!HELLO!'` ? Hint: [2] Answer: [3]

Many methods also take additional parameters between the parentheses, using the more general syntax

object.*method*(parameters)

The first of many such methods we will introduce is `count`:

Syntax for `count`:

[1] `s.lower()`

[2] Use a plus sign to concatenate the pieces.

[3] `s.lower() + s.upper()`

s`.count(`*sub*`)`

Count and return the number of repetitions of a string *sub* that appear as substrings inside the string `s`.

Read and make sure you see the answers are correct:

```
>>> tale = 'This is the best of times.'
>>> tale.count('i')
3
>>> tale.count('is')
2
>>> tale.count('That')
0
>>> tale.count(' ')
5
```

There is a blank between the quotes in the line above. Blanks are characters like any other (except you can't see them)!

Just as the parameter can be replaced by a literal or any expression, the object to which a method is bound with the dot may also be given by a literal, or a variable name, or any expression that evaluates to the right kind of object in its place. This is true for any method call.

Technically the dot between the object and the method name is an operator, and operators have different levels of precedence. It is important to realize that this dot operator has the *highest possible* precedence. *Read* and see the difference parentheses make in the expressions:

```
>>> 'hello ' + 'there'.upper()
'hello THERE'
>>> ('hello ' + 'there').upper()
'HELLO THERE'
```

To see if you understand this precedence, predict the results of each line and then test in the *Shell*:

```
3 * 'X'.count('XXX')
(3 * 'X').count('XXX')
```

There are 0 'XXX's in 'X', but 1 'XXX' in 'XXX'.

Python lets you see all the methods that are bound to an object (and any object of its type) with the built-in function `dir`. To see all string methods, supply the `dir` function with any string. For example, try in the *Shell*:

```
dir('')
```

Many of the names in the list start and end with two underscores, like `__add__`. These are all associated with methods and pieces of data used internally by the Python interpreter. You can ignore them for now. The remaining entries in the list are all user-level methods for strings. You should see `lower` and `upper` among them. Some of the methods are much more commonly used than others.

Object notation

object`.`*method*`(`parameters`)`

has been illustrated so far with just the object type `str`, but it applies to all types. Later in the tutorial methods such as the following will be discussed:

If `seq` is a `list`, `seq.append(element)` appends `element` to the end of the list.

If `myData` is a `file`, `myData.read()` will read and return the entire contents of the file....

2.1.2. String Indices. A string is a sequence of smaller components (individual characters), and it is often useful to deal with parts of strings. Python *indexes* the characters in a string, starting from 0, so for instance, the characters in the string `'computer'` have indices:

character	c	o	m	p	u	t	e	r
index	0	1	2	3	4	5	6	7

Each index is associated with a character, and you reference the individual characters much like in a dictionary. Try the following. (You can skip the comments that make the indices explicit.) Enter in the *Shell*:

```
#    01234567
s = 'computer'
s[0]
s[5]
s[8]
```

You cannot refer directly to a character that is not there. Indices only go to 7 in the example above.

Recall the `len` function, which gives the length of a sequence. It works on strings. Guess the following value, and test in the *Shell*:

```
len(s)
```

A common error is to think the last index will be the same as the length of the string, but as you saw above, that leads to an execution error. If the length of some string is 5, what is the index of its last character? What if the length is 35?

Hopefully you did not count by ones all the way from 0. The indices for a string of length n are the elements of the sequence `range(n)`, which goes from 0 through n-1, or the length of the string minus one, which is 5-1=4 or 35-1 = 34 in these examples.

Sometimes you are interested in the last few elements of a string and do not want to do calculations like this. Python makes it easy. You can index from the *right* end of the string. Since positive integers are used to index from the front, negative integers are used to index from the right end, so the more complete table of indices for `'computer'` gives two alternatives for each character:

character	c	o	m	p	u	t	e	r
index	0	1	2	3	4	5	6	7
index from the right end	-8	-7	-6	-5	-4	-3	-2	-1

Predict and test each individual line, continuing in the *Shell*:

```
s[-1]
s[-3]
s[-10]

it = 'horse'
len(it)
it[-1]
it[1]
```

Be careful - remember what the initial index is!

2.1.3. String Slices. It is also useful to extract larger pieces of a string than a single character. That brings us to *slices*. Try this expression using slice notation, continuing in the *Shell*:

```
s[0:4]
```

Note that s[4] is the first character *past* the slice. The simplest syntax for a slice of a string `s` is:

`s`[startIndex : pastIndex]

This refers to the substring of `s` starting at index `startIndex` and stopping just before index `pastIndex`. It confuses many people that the index after the colon is not the index of the final character in the slice, but that is the system. Predict and try each line individually in the *Shell*:

```
s[2:5]
s[1:3]
```

If you omit the first index, the slice starts from the beginning. If you omit the second index, the slice goes all the way to the end. Predict and try each line individually in the *Shell*:

```
s[:3]
s[5:]
```

Predict and try each line individually in the *Shell*:

```
word = 'program'
word[2:4]
word[1:-3]
```

```
word[3:]
word[3:3]
word[:1] + word[4:]
```

Python evaluates slices in a more forgiving manner than when indexing single characters. In a slice, if you give an index past a limit of where it could be, Python assumes you mean the actual end. Predict and try each line individually in the *Shell*:

```
word[:9]
word[8:10]
```

Enter a slice expression using the variable `word` from above that produces `'gra'`.

A useful string method that uses the ideas of indices and slices is `find`.

Syntax options for `find`:

```
s.find(sub)
s.find(sub, start)
s.find(sub, start, end)
```

Return the integer index in the string *s* of the beginning of first complete occurrence of the substring *sub*. If *sub* does not appear inside s, return -1. The value -1 would be an impossible result if *sub* were found, so if -1 is returned, *sub* must not have been found. If parameters *start* and *end* are not included in the parameter list, the search is through the whole string *s*. If an integer value is given for *start*, the search starts at index *start*. If an integer value is given for *end*, the search ends *before* index *end*. In other words if *start* and *end* appear, then the search is through the slice $s[start : end]$, but the index returned is still counted from the beginning of *s*.

For example, check that the following make sense. The comment line is just there to help you count:

```
>>> #    01234567890
>>> s = 'Mississippi'
>>> s.find('i')
1
>>> s.find('si')
3
>>> s.find('sa')
-1
>>> s.find('si', 4)
6
```

Predict and try each line in the *Shell*:

```
#       0123456789012
line = 'Hello, there!'
line.find('e')
line.find('he')
line.find('e', 10)
line.find('he', 10)
```

We will consider more string methods later, but we can already do useful things with the ones introduced.

Inside the Shell, you can look up documentation on any of the methods listed with the `dir` function. Here is a place that you want to *refer* to the method itself, not *invoke* the method, so note that you get `help` for `s.find` not for `s.find()`. Assuming you defined the string s in the Shell earlier, try in the *Shell*

```
help(s.find)
```

The Python documentation uses square brackets to indicate optional elements that get a default value if you leave them out. This shortens the syntax descriptions.

If you want method documentation when you do not have a variable of the type created, you can also use the type name. Try in the *Shell*:

```
dir(str)
help(str.capitalize)
```

Indexing and slicing works on any kind of Python sequence, so you can index or slice lists also. *Read* this *Shell* session:

```
>>> vals = [5, 7, 9, 22, 6, 8]
>>> vals[1]
7
>>> vals[-2]
6
>>> vals[1:4]
[7, 9, 22]
```

Unlike strings, lists are mutable, as you will see in Section 2.2.1. Indices and slices can also be used in assignment statements to change lists, but in this tutorial we not need list indexing, and we will not discuss this subject further.

2.1.4. Index Variables. All the concrete examples in the last two sections used literal numbers for the indices. That is fine for learning the idea, but in practice, variables or expressions are almost always used for indices. As usual the variable or expression is evaluated before being used. Try in Idle and see that the example program `index1.py` makes sense:

```
s = 'word'
print('The full string is: ', s)
n = len(s)
for i in range(n):
    print()
    print('i =', i)
    print('The letter at index i:', s[i])
    print('The part before index i (if any):', s[:i])
    print('The part before index i+2:', s[:i+2])
```

We will use index variables in more practical situations as we explain more operations with strings.

2.1.5. `split`. Syntax options for the `split` method with a string *s*:

```
s.split()
s.split(sep)
```

The first version splits *s* at any sequence of whitespace (blanks, newlines, tabs) and returns the remaining parts of *s* as a list. If a string *sep* is specified, it is the separator that gets removed from between the parts of the list.

For example, read and follow:

```
>>> tale = 'This is the best of times.'
>>> tale.split()
['This', 'is', 'the', 'best', 'of', 'times.']
>>> s = 'Mississippi'
>>> s.split('i')
['M', 'ss', 'ss', 'pp', '']
>>> s.split()  # no white space
['Mississippi']
```

Predict and test each line in the *Shell*:

```
line = 'Go:  Tear some strings apart!'
seq = line.split()
seq
line.split(':')
line.split('ar')
lines = 'This includes\nsome new\nlines.'
lines.split()
```

2.1.6. `join`. Join is roughly the reverse of split. It joins together a sequence of strings. The syntax is rather different. The separator *sep* comes first, since it has the right type (a string).

Syntax for the `join` method:

```
sep.join(sequence)
```

Return a new string obtained by joining together the *sequence* of strings into one string, interleaving the string *sep* between *sequence* elements.

For example (continuing in the Shell from the previous section, using *seq*):

```
>>> ' '.join(seq)
'Go: Tear some strings apart!'
>>> ''.join(seq)
'Go:Tearsomestringsapart!'
>>> '//'.join(seq)
'Go://Tear//some//strings//apart!'
```

Predict and try each line, continuing in the *Shell*:

```
'##'.join(seq)
':'.join(['one', 'two', 'three'])
```

The methods `split` and `join` are often used in sequence:

EXERCISE 2.1.6.1. * Write a program `underscores.py` that would input a phrase from the user and print out the phrase with the white space between words replaced by an underscore. For instance if the input is "the best one", then it would print "the_best_one". The conversion can be done in one or two statements using the recent string methods.

EXERCISE 2.1.6.2. ** An *acronym* is a string of capital letters formed by taking the first letters from a phrase. For example, SADD is an acronym for 'students against drunk driving'. Note that the acronym should be composed of all capital letters even if the original words are not. Write a program `acronym.py` that has the user input a phrase and then prints the corresponding acronym.

To get you started, here are some things you will need to do. First check that you understand the basic syntax to accomplish the different individual tasks: Indicate the proper syntax using a Python function or operation will allow you to accomplish each task. Invent appropriate variable names for the different parts. This is not complete instructions! the idea is to make sure you know the basic syntax to use in all these situations. See the questions after the list to help you put together the final program.

(1) What type of data will the input be? What type of data will the output be?
(2) Get the phrase from the user.
(3) Convert to upper case.
(4) Divide the phrase into words.
(5) Initialize a new empty list, `letters`.
(6) Get the first letter of each word.
(7) Append the first letter to the list `letters.`
(8) Join the letters together, with no space between them.
(9) Print the acronym.

Which of these steps is in a loop? What `for` statement controls this loop?

Put these ideas together and write and test your program `acronym.py`. Make sure you use names for the objects that are consistent from one line to the next! (You might not have done that when you first considered the syntax and ideas needed for 1-9 above individually.)

2.1.7. Further Exploration: As the `dir('')` list showed, there are many more operations on strings than we have discussed, and there are further variations of the ones above with more parameters. If you want to reach a systematic reference from inside Idle, go to Help-> Python Docs, select Library Reference, and Section 2.3 Built-in Types, and then Section 2.3.6.1, String Methods. (This depends on you being attached to the Internet, or having idle configured to look at a local copy of the official Python documentation.) Many methods use features we have not discussed yet, but currently accessible methods are `capitalize`, `title`, `strip`, `rfind`, `replace`....

2.2. More Classes and Methods

The classes and methods introduced here are all used for the revised mad lib program developed in the next section.

2.2.1. Appending to a List. Before making a version of the madlib program that is much easier to use with new stories, we need a couple of facts about other types of objects that are built into Python.

So far we have used lists, but we have not changed the contents of lists. The most obvious way to change a list is to add a new element onto the end. Lists have the method `append`. It *modifies* the original list. Another word for modifiable is mutable. Lists are *mutable*. Most of the types of object considered so far (int, str, float) are *immutable* or *not* mutable. *Read* and see how the `list` named `words` changes:

```
>>> words = list()
>>> words
[]
>>> words.append('animal')
>>> words
['animal']
>>> words.append('food')
>>> words
['animal', 'food']
>>> words.append('city')
>>> words
['animal', 'food', 'city']
```

This is particularly useful in a loop, where we can accumulate a new list. *Read* the start of this simple example:

```
def multipleAll(numList, multiplier):
    '''Return a new list containing all of the elements of numList,
    each multiplied by multiplier.  For example:

    >>> print(multipleAll([3, 1, 7], 5))
    [15, 5, 35]
    '''
    # more to come
```

Clearly this will be repetitious. We will process each element of the list numList. A for-each loop with numList is appropriate. Also we need to create more and more elements of the new list. The accumulation pattern will work here, with a couple of wrinkles.

Test yourself: If we are going to accumulate a list. How do we initialize the list?

In earlier versions of the accumulation loop, we needed an assignment statement to change the object doing the accumulating, but now the method `append` modifies its list automatically, so we do not need an assignment statement. Read and try the example program `multiply.py`:

```
def multipleAll(numList, multiplier):    #1
    '''Return a new list containing all of the elements of numList,
    each multiplied by multiplier.  For example:

    >>> print(multipleAll([3, 1, 7], 5))
    [15, 5, 35]
    '''

    newList = list()                     #2
    for num in numList:                  #3
        newList.append(num*multiplier)   #4
    return newList                       #5
```

```
print(multipleAll([3, 1, 7], 5))          #6
```

Make sure the result makes sense to you or follow the details of playing computer below.

Line	numList	multiplier	newList	num	comment
1-5	-	-	-	-	definition
6	-	-	-	-	call function
1	[3, 1, 7	5	-		set formal parameters
2	[3, 1, 7	5	[]		
3	[3, 1, 7	5	[]	3	first in list
4	[3, 1, 7	5	[15]	3	append 3*5 = 15
3	[3, 1, 7	5	[15]	1	next in list
4	[3, 1, 7	5	[15, 5]	1	append 1*5 = 5
3	[3, 1, 7	5	[15, 5]	7	last in list
4	[3, 1, 7	5	[15, 5, 35]	7	append 7*5 = 35
3	[3, 1, 7	5	[15, 5, 35]	7	done with list and loop
5	[3, 1, 7	5	[15, 5, 35]	7	return [15, 5, 35]
6	-	-	-	-	print [15, 3, 35]

Using a for-loop and `append` is a powerful and flexible way to derive a new list, but not the only way.[4]

2.2.2. Sets. A list may contain duplicates, as in `[2, 1, 3, 2, 5, 5, 2]`. This is sometimes useful, and sometimes not. You may have learned in math class that a *set* is a collection that does not allow repetitions (a set automatically removes repetitions suggested). Python has a type `set`. Like many type names, it can be used to convert other types. In this case it makes sense to convert any collection, and the process removes duplicates.*Read* and see what happens:

```
>>> numberList = [2, 1, 3, 2, 5, 5, 2]
>>> aSet = set(numberList)
>>> aSet
{1, 2, 3, 5}
```

Set literals are enclosed in braces. Like other collections, a set can be used as a sequence in a `for`-loop. *Read*, and check it makes sense:

```
>>> for item in aSet:
        print(item)



1
2
3
5
```

Predict the result of the following, and then paste it into the *Shell* and test. (Technically, a set is unordered, so you may not guess Python's order, but see if you can get the right length and the right elements in *some* order.)

```
set(['animal', 'food', 'animal', 'food', 'food', 'city'])
```

2.2.3. Constructors. We have now seen several examples of the name of a type being used as a function. *Read* these earlier examples:

```
x = int('123')
s = str(123)
```

[4]There is also a concise syntax called *list comprehension* that allows you to derive a new list from a given sequence. In the example above, we could describe what happens in English as "make newList contain twice each number in numList". This is quite directly translated into an assignment with a list comprehension:

`newList = [2*num for num in numList]`

This is a lot like mathematical set definition notation, except without Greek symbols. List comprehensions also have fancier options, but they are not covered in this tutorial.

```
nums = list()
aSet = set(numberList)
```

In all such cases a new object of the specified type is constructed and returned, and such functions are called *constructors*.

2.3. Mad Libs Revisited

2.3.1. A Function to Ease the Creation of Mad Libs. The versions so far of the Mad Lib program have been fairly easy to edit to contain a different mad lib:

(1) Come up with a new mad lib story as a format string
(2) Produce the list of cues to prompt the user with.

The first is a creative process. The second is a pretty mechanical process of looking at the story string and copying out the embedded cues. The first is best left to humans. The second can be turned over to a Python function to do automatically, as many times as we like, with any story – if we write it once.

Writing the Python code also takes a different sort of creativity! We shall illustrate a creative process. This is a bigger problem than any we have taken on so far. It is hard to illustrate a creative process if the overall problem is too simple.

Try and follow along. *Read* the sample code and pseudocode.

There is nothing to try in the Shell or editor until further notice.

If we follow the last version of the mad lib program, we had a loop iterating through the keys in the story, and making a dictionary entry for each key. The main idea we follow here is to use the format string to automatically generate the sequence of keys. Let us plan this unified task as a new function:

```
def getKeys(formatString):
    '''formatString is a format string with embedded dictionary keys.
    Return a list containing all the keys from the format string.'''
    # more to come
```

The keys we want are embedded like `{animal}`. There may be any number of them in the format string. This indeterminacy suggests a loop to extract them. At this point we have only considered `for`-loops. There is no obvious useful sequence to iterate through in the loop (we are trying to *create* such a sequence). The only pattern we have discussed that does not actively process each element of a significant list is a repeat-loop, where we just use the loop to repeat the correct number of times. This will work in this case.

First: how many times do we want to pull out a key – once for each embedded format. So how do we count those?

The `count` method is obviously a way to count. However we must count a fixed string, and the whole embedded formats vary (with different keys in the middle. A common part is `'{'`, and this should not appear in the regular text of the story, so it will serve our purpose:

```
repetitions = formatString.count('{')
for i in range(repetitions):
    ...
```

This is certainly the most challenging code to date. Before jumping into writing it all precisely, we can give an overall plan in pseudo-code. For a plan we need an idea of what quantities we are keeping track of, and name them, and outline the sequence of operations with them.

Think about data to name:

In this case we are trying to find a list. We will need to extract one element at a time and add it to the list, so we need a list, say `keyList`.

The central task is to identifying the individual keys. When we find a key we can call it `key`.

Think about identifying the text of individual keys. This may be too hard to think of in the abstract, so let us use as a concrete example, and let us keep it simple for the moment. Suppose the data in `formatString` starts off as follows. The lines with numbers are added to help us refer to the indices. *Display* of possible data:

```
#           111111111122222222223333333
# 01234567890123456789012345678901234567
 'blah {animal} blah blah {food} ...'
```

The first key is `'animal'` at `formatString[6:12]`. The next key is `'food'` at `formatString[25:29]`. To identify each key as part of `formatString` we need not only the variable `formatString`, but also index variables to locate the start and end of the slices. Obvious names for the indices are `start` and `end`. We want to keep them current so the next key slice will always be

```
key = formatString[start : end]
```

Let us now put this all in an overall plan. We will have to continuously modify the start and end indices, the key, and the list. We have a basic pattern for accumulating a list, involving initializing it and appending to it. We can organize a plan, partly fleshed out, with a couple of approximations to be worked out still. The parts that are not yet in Python are emphasized:

```
def getKeys(formatString):

    keyList = list()
    ?? other initializations ??
    repetitions = formatString.count('{')
    for i in range(repetitions):
        find the start and end of the next key
        key = formatString[start : end]
        keyList.append(key)

    return keyList
```

We can see that the main piece left is to find the start and end indices for each key. The important word is *find*: the method we consider is `find`. As with the plan for using `count` above, the beginnings of keys are identified by the specific string `'{'`. We can look first at

```
formatString.find('{')
```

but that is not the full solution. If we look at our concrete example, the value returned is 5, not 6. How in general would we locate the beginning of the slice we want?

We do not want the position of the *beginning* of `'{'`, but the position just *after* the `'{'`. Since the length of `'{'` is 1, the correct position is 5+1 = 6. We can generalize this to

```
start = formatString.find('{') + 1
```

OK, what about `end`? Clearly it is at the `'}'`. In this example,

```
formatString.find('}')
```

gives us 12, exactly the right place for the end of the slice (one place past the actual end).

There is a subtle issue here that will be even more important later: We will kep wanting to find the *next* brace,a dn not keep finding the *first* brace. How do we fix that?

Recall there was an alternate syntax for `find`, specifying the first place to search! That is what we need. Where should we start? Well, the end must come after the start of the key, our variable `start`:

```
start = formatString.find('{') + 1
end = formatString.find('}', start)
```

Figuring out how to find the first key is important, but we are not home free yet. We need to come up with code that works in a loop for the *later* keys. This code will not work for the next one. Why?

The search for `'{'` will again start from the beginning of the format string, and will find the first key again. So what code will work for the *second* search? We search for the start of the next key going from the end of the last one:

```
start = formatString.find('{', end) + 1
end = formatString.find('}', start)
```

This code will also work for later times through the loop: each time uses the `end` from the previous time through the loop.

So now what do we do for finding the first key? We could separate the treatment of the first key from all the others, but an easier approach would be to see if we can use the same code that already works for the later repetitions, and initialize variables right to make it work. If we are to find the *first* key with

```
start = formatString.find('{', end) + 1
```

then what do we need? Clearly `end` needs to have a value. (There will not be a *previous* loop to give it a value.) What value should we initialize it to? The first search starts from the beginning of the string at index 0, so the full code for this function is

```
def getKeys(formatString):
    '''formatString is a format string with embedded dictionary keys.
    Return a list containing all the keys from the format string.'''

    keyList = list()
    end = 0
    repetitions = formatString.count('{')
    for i in range(repetitions):
        start = formatString.find('{', end) + 1
        end = formatString.find('}', start)
        key = formatString[start : end]
        keyList.append(key)

    return keyList
```

Look the code over and see that it makes sense. See how we continuously modify `start`, `end`, `key`, and `keyList`. Since we have coded this new part as a function, it is easy to test without running a whole revised mad lib program. We can just run this function on some test data, like the original story, and see what it does. Run the example program `testGetKeys.py`:

```
def getKeys(formatString):
    '''formatString is a format string with embedded dictionary keys.
    Return a list containing all the keys from the format string.'''

    keyList = list()
    end = 0
    repetitions = formatString.count('{')
    for i in range(repetitions):
        start = formatString.find('{', end) + 1
        end = formatString.find('}', start)
        key = formatString[start : end]
        keyList.append(key)

    return keyList

originalStory = """
Once upon a time, deep in an ancient jungle,
there lived a {animal}.  This {animal}
liked to eat {food}, but the jungle had
very little {food} to offer.  One day, an
explorer found the {animal} and discovered
it liked {food}.  The explorer took the
{animal} back to {city}, where it could
eat as much {food} as it wanted.  However,
the {animal} became homesick, so the
explorer brought it back to the jungle,
leaving a large supply of {food}.

The End
"""

print(getKeys(originalStory))
```

The functions should behave as advertised.

Look back on the process described to come up with the `getKeys` function. One way of approaching the creative process of coding this function was provided. There are many other results and approaches possible, but the discussion did illustrate a number of useful ideas which you might adapt to other problems, in different orders and proportions, that are summarized in the next section.

2.3.2. Creative Problem Solving Steps.

- Clearly define the problem. Encapsulating the problem in a function is useful, with inputs as parameters and results returned. Include a complete documentation string, and a clear example (or examples) of what it is to do.
- If the problem is too complicated to just solve easily, straight away, it is often useful to construct a representative *concrete* case and write down *concrete* steps appropriate to this problem.
- Think of the data in the problem, and give names to the pieces you will need to refer to. Clearly identify the ideas that the names correspond to. When using sequences like lists or strings, you generally need names not only for the whole collection, but also parts like items and characters or substrings, and often indices that locate parts of the collection.
- Plan the overall approach to the problem using a mixture of Python and suggestive phrases (called pseudo-code). The idea is to refine it to a place where you can fairly easily figure how to replace the phases with Python.
- Replace your pseudo-code parts with Python. If you had a concrete example to guide, you may want one of more further concrete examples with different specific data, to make sure you come up with code for a generalization that works in all cases.
- Recognize where something is being repeated over and over, and think how to structure appropriate loops. Can you incorporate any patterns you have seen before?
- If you need to create a successive modification loop, think of how to approach the first repetition and then how to modify the data for the later times through the loop. Usually you can make the first time through the loop fit the more general pattern needed for the repetitions by making appropriate initializations before the loop.
- Check and test your code, and correct as necessary.

2.3.3. The Revised Mad Lib Program. There is still an issue for use of `getKeys` in the mad lib program: the returned list has repetitions in it, that we do not want in the mad lib program. We can easily create a collection without repetitions, how?

One approach is to make a `set` from the list returned. A neater approach would be to just have the getKeys function return a `set` in the first place. We need to slightly change to `getKeys`' documentation string and the final `return` line. This will be included in a new version of the mad lib program, which makes it easy to substitute a new story. We will make the story's format string be a parameter to the central method, `tellStory`. We will also put the clearly identified step of filling the dictionary with the user's picks in a separate function. We will test `tellStory` with the original story. Note the changes included in `madlib2.py` and run:

```
"""
madlib2.py
Interactive display of a mad lib, which is provided as a Python format string,
with all the cues being dictionary formats, in the form {cue}.
In this version, the cues are extracted from the story automatically,
and the user is prompted for the replacements.

Original mad lib adapted from code of Kirby Urner

"""

def getKeys(formatString):   # change: returns a set
    '''formatString is a format string with embedded dictionary keys.
    Return a set containing all the keys from the format string.'''
    keyList = list()
```

```
    end = 0
    repetitions = formatString.count('{')
    for i in range(repetitions):
        start = formatString.find('{', end) + 1
        end = formatString.find('}', start)
        key = formatString[start : end]
        keyList.append(key) # may add duplicates
    return set(keyList) # removes duplicates: no duplicates in a set

def addPick(cue, dictionary): # from madlib.py
    '''Prompt the user and add one cue to the dictionary.'''
    prompt = 'Enter an example for ' + cue + ': '
    dictionary[cue] = input(prompt)

def getUserPicks(cues):
    '''Loop through the collection of cue keys and get user choices.
    Return the resulting dictionary.
    '''
    userPicks = dict()
    for cue in cues:
        addPick(cue, userPicks)
    return userPicks

def tellStory(story):
    '''story is a format string with Python dictionary references embedded,
    in the form {cue}.  Prompt the user for the mad lib substitutions
    and then print the resulting story with the substitutions.
    '''

    cues = getKeys(story)
    userPicks = getUserPicks(cues)
    print(story.format(**userPicks))

def main():
    originalStory = '''
Once upon a time, deep in an ancient jungle,
there lived a {animal}.  This {animal}
liked to eat {food}, but the jungle had
very little {food} to offer.  One day, an
explorer found the {animal} and discovered
it liked {food}.  The explorer took the
{animal} back to {city}, where it could
eat as much {food} as it wanted.  However,
the {animal} became homesick, so the
explorer brought it back to the jungle,
leaving a large supply of {food}.

The End
'''
    tellStory(originalStory)


main()
```

Does the use of well-named functions make it easier to follow this code? Make sure you follow the flow of execution and data.

After Python file manipulation is introduced, in Exercise 2.5.0.3you can modify the program to work on a madlib format string chosen by the user and taken from a file.

EXERCISE 2.3.3.1. ** Rename the example file locationsStub.py to be locations.py, and complete the function printLocations, to print the index of *each* location in the string `s` where `target` is located. For example, `printLocations('This is a dish', 'is')` would go through the string `'This is a dish'` looking for the index of places where `'is'` appears, and would return `[2, 5, 11]`. Similarly `printLocations('This is a dish', 'h')` would return `[1, 13]`. The program stub already uses the string method `count`. You will need to add code using the more general form of `find`.

2.4. Graphics

Graphics make programming more fun for many people. To fully introduce graphics would involve many ideas that would be a distraction now. This section introduces a simplified graphics module developed by John Zelle for use with his Python Programming book. My slight elaboration of his package is graphics.py in the example programs.

Repeated caution: In Windows XP or Vista, be sure to start Idle from the shortcut provided in the examples (in the same directory as graphics.py). Do *not* start by clicking on an existing file to get a context menu and choosing Open With Idle: The 'Open With Idle' allows you to edit, but then when you go to run your graphics program, it fails miserably and with no clear reason.

2.4.1. A Graphics Introduction in the Shell. Make sure you have Idle started from inside your Python folder, and have graphics.py in the *same* folder, so the Python interpreter can find it.

Note: you will just be a *user* of the graphics.py code, so you do *not* need to understand the inner workings! It uses all sorts of features of Python that are way beyond these tutorials. There is *no particular need* to open graphics.py in the Idle editor.

You will definitely want to be a *user* of the graphical module.

In Idle, in the *Shell*, enter the following lines, one at a time and read the explanations:

```
from graphics import *
```

Zelle's graphics are not a part of the standard Python distribution. For the Python interpreter to find Zelle's module, it must be *imported*. The line above makes all the types of object of Zelle's module accessible, as if they were already defined like built-in types `str` or `list`.

Pause after you enter the *opening* parenthesis below:

```
win = GraphWin(
```

The Idle editor tries to help you by displaying a pop-up tool tip with the parameter names and sometimes a default value after an equal sign. The *default value* is used if you supply nothing. In this case we will use the default values, so you can just finish by entering the closing parenthesis, now, completing

```
win = GraphWin()
```

Look around on your screen, and possibly underneath other windows: There should be a new window labeled "Graphics Window". Bring it to the top, and preferably drag it around to make it visible beside your Shell window. A `GraphWin` is a type of object from Zelle's graphics package that automatically displays a window when it is created. The assignment statement remembers the window object as `win` for future reference. (This will be our standard name for our graphics window object.) A small window, 200 by 200 *pixels* is created. A *pixel* is the smallest little square that can by displayed on your screen. Modern screen usually have more than 1000 pixels across the whole screen.

Again, pause after entering the *opening* parenthesis below, and see how Idle hints at the meaning of the parameters to create a `Point` object. Then complete the line as given below:

```
pt = Point(100, 50)
```

This creates a `Point` object and assigns it the name `pt`. Unlike when a `GraphWin` is created, nothing is immediately displayed: In theory you could have more than one `GraphWin`. Zelle designed the graphics

module so you must tell Python into *which* `GraphWin` to draw the `Point`. A `Point` object, like each of the graphical objects that can be drawn on a `GraphWin`, has a method[5] `draw`. Enter

```
pt.draw(win)
```

Now you should see the `Point` if you look hard in the Graphics Window - it shows as a single, small, black pixel. Graphics windows have a Cartesian (x,y) coordinate system. The dimensions are initially measured in pixels. The first coordinate is the horizontal coordinate, measured from left to right, so 100 is about half way across the 200 pixel wide window. The second coordinate, for the vertical direction, increases going *down* from the top of the window by default, not *up* as you are likely to expect from geometry or algebra class. The coordinate 50 out of the total 200 vertically should be about 1/4 of the way *down* from the top. We will see later that we can reorient the coordinate system to fit our taste.

Enter both of the lines of code

```
cir = Circle(pt, 25)
cir.draw(win)
```

The first line creates a `Circle` object with center at the previously defined `pt` and with radius 25. This object is remembered with the name `cir`. As with all graphics objects that may be drawn within a `GraphWin`, it is only made visible by explicitly using its `draw` method:

So far, everything has been drawn in the default color black. Graphics objects like a Circle have methods to change their colors. Basic color name strings are recognized. You can choose the color for the circle outline as well as filling in the inside. Enter both lines

```
cir.setOutline("red")
cir.setFill("blue")
```

Now add

```
line = Line(pt, Point(150, 100))
line.draw(win)
```

A Line object is constructed with two Points as parameters. In this case we use the previously named Point, `pt`, and specify another Point directly. Technically the Line object is a *segment* between the the two points.

A rectangle is also specified by two points. The points must be diagonally opposite corners. Try

```
rect = Rectangle(Point(20, 10), pt)
rect.draw(win)
```

You can move objects around in a GraphWin. This will be handy for animation, shortly. The parameters to the `move` method are the amount to shift the x and y coordinates. See if you can guess the result before you enter:

```
line.move(10, 40)
```

Did you remember that the y coordinate increases *down* the screen?

Feel free to further modify the graphics window using the methods introduced. To do this in a more systematic and easily reproduced way, we will shortly switch to program files, but then you do not get to see the effect of each statement individually and immediately!

Take your last look at the Graphics Window, and make sure that all the steps make sense. Then destroy the window `win` with the `GraphWin` method `close`:

```
win.close()
```

An addition I have made to Zelle's package is the ability to print a string value of graphics objects for debugging purposes. Assuming you downloaded graphics.py from the hands-on site (not Zelle's), continue in the *Shell* with

```
print(line)
```

If some graphics object isn't visible because it is underneath something else of off the screen, this sort of output might be a good reality check.

[5] The basic ideas of objects and methods were introduced in Section 2.1.1.

2.4.2. Sample Graphics Programs. Here is a very simple program, `face.py`. The only interaction is to click the mouse to close the graphics window. Have a directory window open to the Python examples folder containing `face.py`. In Windows you can double click on the icon for `face.py` to run it.

After you have checked out the picture, click with the mouse inside the picture, as requested, to terminate the program.

After you have run the program, you can examine the program in Idle or look below. The whole program is shown first; smaller pieces of it are discussed later:

```
'''A simple graphics example constructs a face from basic shapes.
'''

from graphics import *

def main():
    winWidth = 200  # give a name to the window width
    winHeight = 150 #    and height
    win = GraphWin('Face', winWidth, winHeight) # give title and dimensions
    win.setCoords(0, 0, winWidth, winHeight) # make right side up coordinates!

    head = Circle(Point(40,100), 25) # set center and radius
    head.setFill("yellow")
    head.draw(win)

    eye1 = Circle(Point(30, 105), 5)
    eye1.setFill('blue')
    eye1.draw(win)

    eye2 = Line(Point(45, 105), Point(55, 105)) # set endpoints
    eye2.setWidth(3)
    eye2.draw(win)

    mouth = Oval(Point(30, 90), Point(50, 85)) # set corners of bounding box
    mouth.setFill("red")
    mouth.draw(win)

    message = Text(Point(winWidth/2, 20), 'Click anywhere to quit.')
    message.draw(win)
    win.getMouse()
    win.close()

main()
```

Let us look at individual parts.

Until further notice the set-off code is for you to read and have explained.

Immediately after the documentation string, always have the import line in your graphics program, to allow easy access to the graphics.py module:

```
from graphics import *
```

Though not a graphics issue, the first two lines of the `main` method illustrate a very good practice:

```
    winWidth = 200  # give a name to the window width
    winHeight = 150 #    and height
```

Important parameters for your programs should get names. Within the program the names will make more sense to the human user than the literal data values. Plus, in this program, these parameters are used several times. If I choose to change the window size to 400 by 350, I only need to change the value of each dimension in *one* place!

```
    win = GraphWin('Face', winWidth, winHeight) # give title and dimensions
        win.setCoords(0, 0, winWidth, winHeight) # make right side up coordinates!
```

The first line shows the more general parameters for constructing a new GraphWin, a window title and dimensions in pixels. The second line shows how to turn the coordinate system right-side-up, so the y coordinate increases *up* the screen. The setCoords method sets up a new coordinate system, where the first two numbers are the coordinates you wish to use for the lower left corner of the window, and the last two numbers are the coordinates of the upper right corner. Thereafter, all coordinates are given in the new coordinate system, and the graphics module silently calculates the correct underlying pixel positions. All the lines of code up to this point in the program are my *standard* graphics program starting lines (other than the specific values for the title and dimensions). You will likely start your programs with similar code.

```
        head = Circle(Point(40,100), 25) # set center and radius
        head.setFill("yellow")
        head.draw(win)

        eye1 = Circle(Point(30, 105), 5)
        eye1.setFill('blue')
        eye1.draw(win)
```

The lines above create two circles, in each case specifying the centers directly. They are filled in and made visible.

```
        eye2 = Line(Point(45, 105), Point(55, 105)) # set endpoints
        eye2.setWidth(3)
        eye2.draw(win)
```

The code above draws and displays a line, and illustrates another method available to graphics object, `setWidth`, making a *thicker* line.

```
    mouth = Oval(Point(30, 90), Point(50, 85)) # set corners of bounding box
        mouth.setFill("red")
        mouth.draw(win)
```

The code above illustrates another kind of graphics object, an Oval (or ellipse). There are several ways an oval could be specified. Zelle chose to have you specify the corners of the bounding box that is just as high and as wide as the oval. This rectangle is only imagined, not actually drawn. (If you want to see such a rectangle, create a Rectangle object with the same two Points as paramaeters..)

The exact coordinates for the parts were determined by a number of trial-and-error refinements to the program. An advantage of graphics is that you can *see* the results of your programming, and make changes if you do not like the results!

The final action is to have the user signal to close the window. Just as with waiting for keyboard input from `input` or `input`, it is important to *prompt* the user before waiting for a response! In a GraphWin, the prompt must be made with a `Text` object displayed explicitly before the response is expected. Lines like the following will often end a program that has a final displayed picture:

```
        message = Text(Point(winWidth/2, 20), 'Click anywhere to quit.')
        message.draw(win)
        win.getMouse()
        win.close()
```

The parameters to construct the Text object are the point at the *center* of the text, and the text string itself. See how the text position is set up to be centered from left to right, half way across the window's width. Also note, that because the earlier `win.setCoord` call put the coordinates in the normal orientation, the y coordinate, 20, is close to the bottom of the window.

After the first two lines draw the promping text, the line `win.getMouse()` *waits* for a mouse click. In this program, the position is not important. (In the next example the position of this mouse click will be used.) As you have seen before, `win.close()` closes the graphics window.

While our earlier text-based Python programs have automatically terminated after the last line finishes executing, that is not true for programs that create new windows: The graphics window must be explicitly closed. The `win.close()` is necessary.

You can copy the form of this program for other simple programs that just draw a picture. The size and title on the window will change, as well as the specific graphical objects, positions, and colors. Something like the last four lines can be used to terminate the program.

Another simple drawing example is `balloons.py`. Feel free to run it and look at the code in Idle. Note that the steps for the creation of all three balloons are identical, except for the location of the center of each balloon, so a loop over a list of the centers makes sense.

The next example, `triangle.py`, illustrates similar starting and ending code. In addition it explicitly interacts with the user. Rather than the code specifying literal coordinates for all graphical objects, the program remembers the places where the *user* clicks the mouse, and uses them as the vertices of a triangle.

Return to the directory window for the Python examples. In Windows you can double click on the icon for `triangle.py` to run it.

While running the program, follow the prompts in the graphics window and click with the mouse as requested.

After you have run the program, you can examine the program in Idle or look below:

```
'''Program: triangle.py or triangle.pyw (best name for Windows)
Interactive graphics program to draw a triangle,
with prompts in a Text object and feedback via mouse clicks.
Illustrates all of the most common GraphWin methods, plus
some of the ways to change the appearance of the graphics.
'''

from graphics import *

def main():
    winWidth = 300
    winHeight = 300
    win = GraphWin('Draw a Triangle', winWidth, winHeight)
    win.setCoords(0, 0, winWidth, winHeight) # make right-side-up coordinates!
    win.setBackground('yellow')

    message = Text(Point(winWidth/2, 20), 'Click on three points')
    message.draw(win)

    # Get and draw three vertices of triangle
    p1 = win.getMouse()
    p1.draw(win)

    p2 = win.getMouse()
    p2.draw(win)

    p3 = win.getMouse()
    p3.draw(win)

    vertices = [p1, p2, p3]
    triangle = Polygon(vertices)
    triangle.setFill('gray')
    triangle.setOutline('cyan')
    triangle.setWidth(4) # width of boundary line
    triangle.draw(win)

    # Wait for a final click to exit
```

```
        message.setText('Click anywhere to quit.')
        message.setTextColor('red')
        message.setStyle('italic')
        message.setSize(20)

        win.getMouse()
        win.close()

    main()
```

Let us look at individual parts.

Until further notice the set-off code is for you to read and have explained.

The lines before

```
win.setBackground('yellow')
```

are standard starting lines (except for the specific values chosen for the width, height, and title). The background color is a property of the whole graphics window that you can set. The line above illustrates the last new common method of a `GraphWin`.

```
message = Text(Point(winWidth/2, 20), 'Click on three points')
message.draw(win)
```

Again a Text object is created. We will see below how the `Text` object can be modified later. This is the prompt for user action, expected for use in the lines

```
# Get and draw three vertices of triangle
p1 = win.getMouse()
p1.draw(win)

p2 = win.getMouse()
p2.draw(win)

p3 = win.getMouse()
p3.draw(win)
```

The `win.getMouse()` method (with no parameters), waits for you to click the mouse inside `win`. Then the `Point` where the mouse was clicked is *returned*. In this code three mouse clicks are waited for, remembered in variables `p1`, `p2`, and `p3`, and the points are drawn.

Next we introduce a very versatile type of graphical object, a `Polygon`, which may have any number of vertices specified in a list as its parameter. We see that the methods `setFill` and `setOutline` that we used earlier on a `Circle`, and the `setWidth` method available for a `Line`, also apply to a `Polygon`, (and also to other graphics objects).

```
vertices = [p1, p2, p3]
triangle = Polygon(vertices)
triangle.setFill('gray')
triangle.setOutline('cyan')
triangle.setWidth(4)
triangle.draw(win)
```

The next few lines illustrate most of the ways a Text object may be modified. Not only may the text string be changed. The appearance may be changed like in most word processors. The reference pages for graphics.py give the details.

```
# Wait for a final click to exit
 message.setText('Click anywhere to quit.')
 message.setTextColor('red')
 message.setStyle('italic')
 message.setSize(20)
```

After this prompt (with its artificially elaborate styling), we have the standard finishing lines:

```
win.getMouse()
win.close()
```

2.4.3. A Windows Operating System Specialization: .pyw. This Windows-specific section is not essential. It does describe how to make some Windows graphical programs run with less clutter.

If you ran the triangle.py program by double clicking its icon under Windows, you might have noticed a console window first appearing, followed by the graphics window. For this program, there was no keyboard input or screen output through the console window, so the console window was unused and unnecessary. In such cases, under Windows, you can change the source file extension from .py to .pyw, suppressing the display of the console window. If you are using windows, check it out.

The distinction is irrelevant inside Idle, which always has its Shell window.

2.4.4. Graphics.py vs. Event Driven Graphics. This optional section only looks forward to more elaborate graphics systems than are used in this tutorial.

One limitation of the graphics.py module is that it is not robust if a graphics window is closed by clicking on the standard operating system close button on the title bar. If you close a graphics window that way, you are likely to get a Python error message. On the other hand, if your program creates a graphics window and then terminates abnormally due to some *other* error, the graphics window may be left orphaned. In this case the close button on the title bar is important: it is the easiest method to clean up and get rid of the window!

This lack of robustness is tied to the simplification designed into the graphics module. Modern graphics environments are *event driven*. The program can be interrupted by input from many sources including mouse clicks and key presses. This style of programming has a considerable learning curve. In Zelle's graphics package, the complexities of the event driven model are pretty well hidden. If the programmer wants user input, only one type can be specified at a time (either a mouse click in the graphics window via the getMouse method, or via the `input` or `input` keyboard entry methods into the Shell window).

2.4.5. The Documentation for graphics.py. Thus far various parts of Zelle's graphics package have been introduced by example. A systematic reference to Zelle's graphics package with the form of all function calls is at `http://mcsp.wartburg.edu/zelle/python/graphics/graphics/index.html`. We have introduced most of the important concepts and methods.

One special graphics input object type, `Entry`, will be discussed later. You might skip it for now. Another section of the reference that will not be pursued in the tutorials is the `Image` class.

Meanwhile you can look at `http://mcsp.wartburg.edu/zelle/python/graphics/graphics/index.html`. It is important to pay attention to the organization of the reference: Most graphics object share a number of common methods. Those methods are described together, first. Then, under the headings for specific types, only the specialized additional methods are discussed.

EXERCISE 2.4.5.1. * Make a program `scene.py` creating a scene with the graphics methods. You are likely to need to adjust the positions of objects by trial and error until you get the positions you want. *Make sure you have graphics.py in the same directory as your program.*

EXERCISE 2.4.5.2. * Elaborate your scene program so it becomes `changeScene.py`, and changes one or more times when you click the mouse (and use `win.getMouse()`). You may use the position of the mouse click to affect the result, or it may just indicate you are ready to go on to the next view.

2.4.6. Issues with Mutable Objects: The Case for `clone`. Zelle chose to have the constructor for a Rectangle take diagonally opposite corner points as parameters. Suppose you prefer to specify only one corner and also specify the width and height of the rectangle. You might come up with the following function, `makeRect`, to return such a new Rectangle. *Read* the following attempt:

```
def makeRect(corner, width, height):
    '''Return a new Rectangle given one corner Point and the dimensions.'''
    corner2 = corner
    corner2.move(width, height)
    return Rectangle(corner, corner2)
```

The second corner must be created to use in the `Rectangle` constructor, and it is done above in two steps. Start `corner2` from the given `corner` and shift it by the dimensions of the Rectangle to the other corner. With both corners specified, you can use Zelle's version of the Rectangle constructor.

Unfortunately this is an incorrect argument. Run the example program makeRectBad.py:

```
'''Program: makeRectBad.py
Attempt a function makeRect (incorrectly), which takes
a takes a corner Point and dimensions to construct a Rectangle.
'''

from graphics import *

def makeRect(corner, width, height):  # Incorrect!
    '''Return a new Rectangle given one corner Point and the dimensions.'''
    corner2 = corner
    corner2.move(width, height)
    return Rectangle(corner, corner2)

def main():
    winWidth = 300
    winHeight = 300
    win = GraphWin('Draw a Rectangle (NOT!)', winWidth, winHeight)
    win.setCoords(0, 0, winWidth, winHeight)
    rect = makeRect(Point(20, 50), 250, 200)
    rect.draw(win)

    # Wait for another click to exit
    msg = Text(Point(winWidth/2, 20),'Click anywhere to quit.')
    msg.draw(win)
    win.getMouse()
    win.close()

main()
```

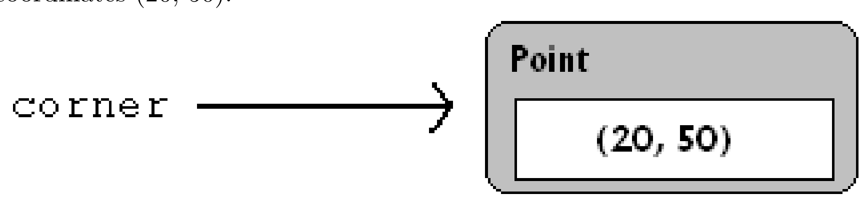
By stated design, this program should draw a rectangle with one corner at the point (20, 50) and the other corner at (20+250, 50+200) or the point (270, 250), and so the rectangle should take up most of the 300 by 300 window. When you run it however that is not what you see. Look carefully. You should just see one `Point` toward the upper right corner, where the second corner *should* be. Since a `Rectangle` was being drawn, it looks like it is the tiniest of `Rectangle`s, where the opposite corners are at the same point! Hm, well the program *did* make the corners be the same *initially*. *Recall* we set

```
corner2 = corner
```

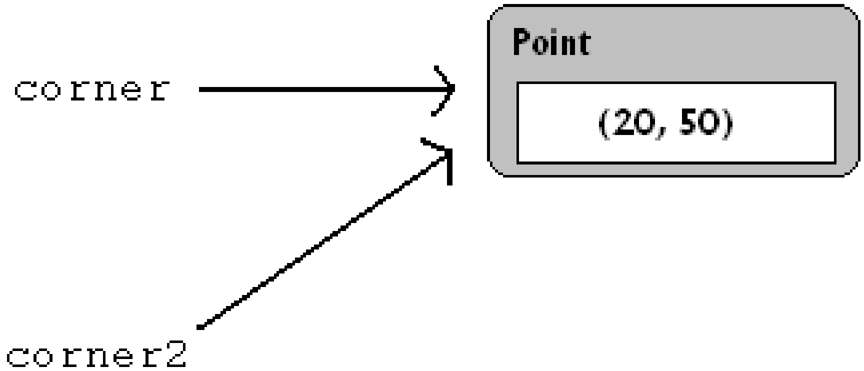
What happens after that?

Read and follow the details of what happens.

We need to take a much more careful look at what naming an object means. A good way to visualize this association between a name and an object is to draw an arrow from the name to the object associated with it. The object here is a `Point`, which has an x and y coordinate describing its state, so when the `makeRect` method is started the parameter name `corner` is associated with the actual parameter, a `Point` with coordinates (20, 50).



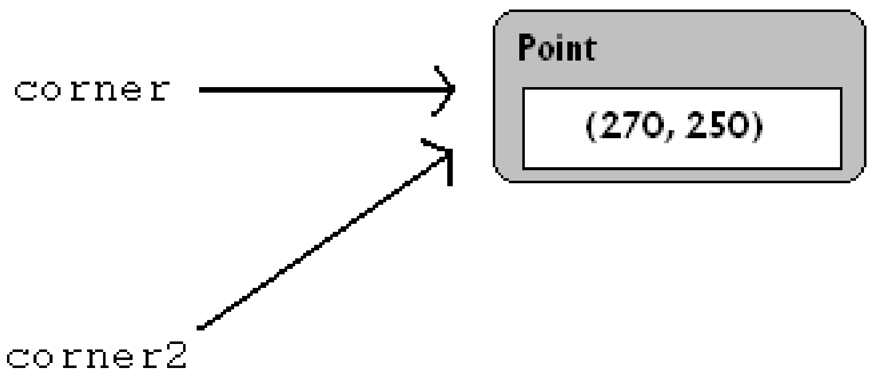
Next, the assignment statement associates the name `corner2` with the same object. It is another name, or *alias*, for the original `Point`.



The next line,

```
corner2.move(width, height)
```

internally changes or *mutates* the `Point` object, and since in this case width is 250 and height is 200, the coordinates of the `Point` associated with the name `corner2` change to 20+250=270 and 50+200=250:



Look! The name `corner` is still associated with the same object, but that object has changed internally! That is the problem: we wanted to keep the name `corner` associated with the point with *original* coordinates, but it has been *modified*.

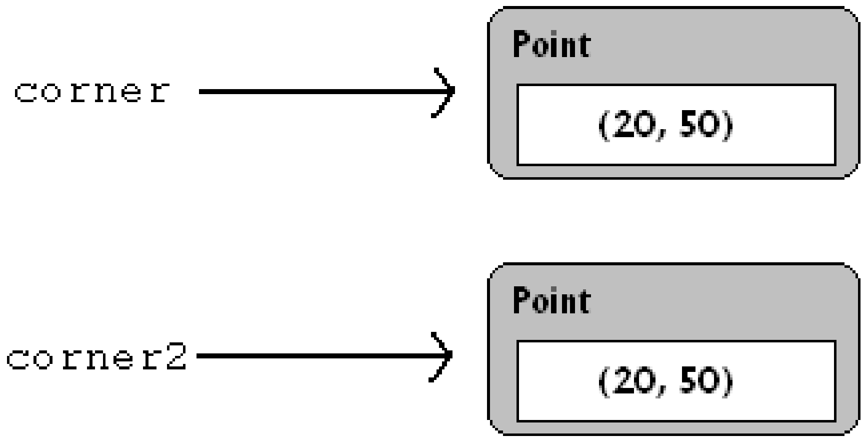
The solution is to use the `clone` method that is defined for all the graphical objects in graphics.py. It creates a *separate* object, which is a copy with an equivalent state. We just need to change the line

```
corner2 = corner
```

to

```
corner2 = corner.clone()
```

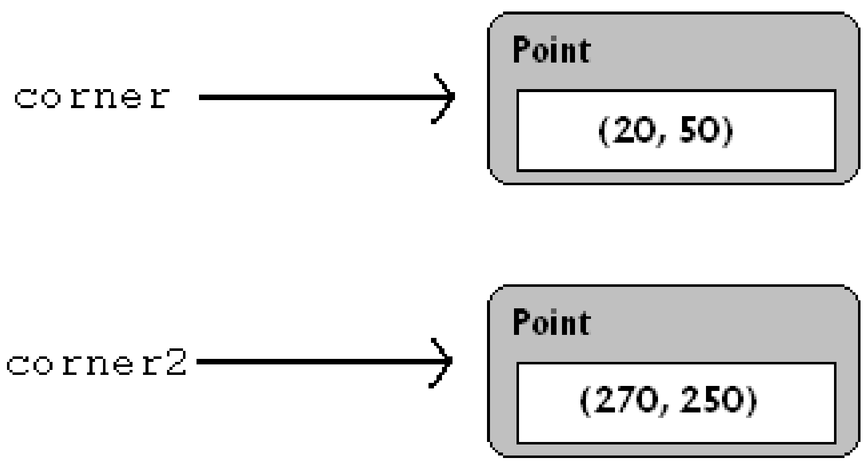
A diagram of the situation after the cloning is:



Though `corner` and `corner2` refer to points with equivalent coordinates, they do not refer to the same *object*. Then after

```
corner2.move(width, height)
```

we get:



No conflict: `corner` and `corner2` refer to the corners we want. Run the corrected example program, makeRectange.py.

2.4.7. More on Mutable and Immutable Types. Read this section if you want a deeper understanding of the significance of muable and immutable objects.

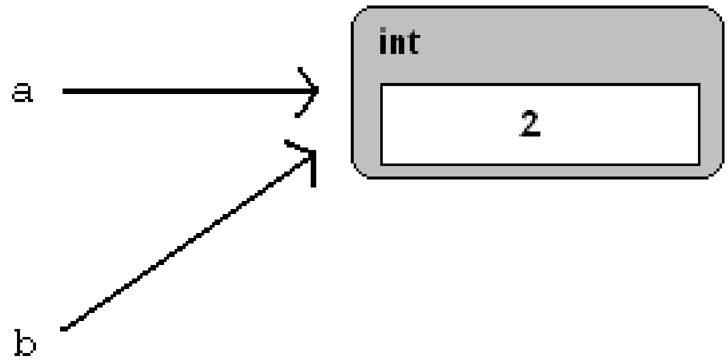
This alias problem only came up because a `Point` is mutable. We had no such problems with the immutable types `int` or `str`.

Read and follow the discussion of the following code.

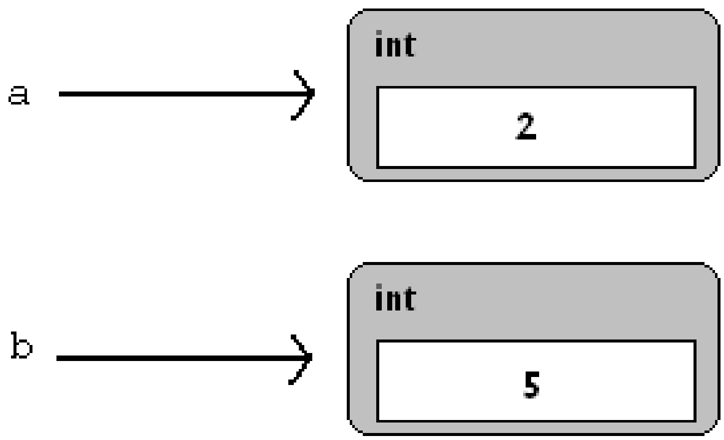
Just for comparison, consider the corresponding diagrams for code with `int`s that looks superficially similar:

```
a = 2
b = a
b = b + 3
```

After the first two lines we have an alias again:



The third line does not change the int object 2. The result of the addition operation refers to a *different* object, 5, and the name `b` is assigned to it:



Hence `a` is still associated with the integer 2 – no conflict.

It is not technically correct to think of `b` as *being* the number 2, and then 5, but a little sloppiness of thought does not get you in trouble with *immutable* types. With mutable types, however, be very careful of aliases. Then it is very important to remember the indirectness: that a name is not the same thing as the object it refers to.

Another mutable type is `list`. A list can be cloned with the slice notation: `[:]`. Try the following in the *Shell*:[6]

```
nums1 = [1, 2, 3]
nums2 = nums1[:]
nums2.append(4)
nums1
nums2
```

2.4.8. Animation. Run the example program, backAndForth0.py. The whole program is shown below for convenience. Then each individual new part of the code is discussed individually:

```
'''Test animation and depth.
'''
from graphics import *
import time

def main():
    winWidth = 300
```

[6]Actually, lists are even trickier, because the elements of a `list` are arbitrary: There can still be issues of dependence between the original and cloned list if the *elements* of the list are themselves mutable, and then you choose to mutate an element.

```
    winHeight = 300
    win = GraphWin('Back and Forth', winWidth, winHeight)
    win.setCoords(0, 0, winWidth, winHeight)

    rect = Rectangle(Point(200, 90), Point(220, 100))
    rect.setFill("blue")
    rect.draw(win)

    cir1 = Circle(Point(40,100), 25)
    cir1.setFill("yellow")
    cir1.draw(win)

    cir2 = Circle(Point(150,125), 25)
    cir2.setFill("red")
    cir2.draw(win)

    for i in range(46):  # animate cir1 to the right
        cir1.move(5, 0)
        time.sleep(.05)
    for i in range(46):  # animate cir1 to the left
        cir1.move(-5, 0)
        time.sleep(.05)

    # Wait for a final click to exit
    Text(Point(winWidth/2, 20), 'Click anywhere to quit.').draw(win)
    win.getMouse()
    win.close()

main()
```

Read the discussion below of pieces of the code from the program above. Do not try to execute fragments alone.

There is a new form of `import` statement:

```
from graphics import *
import time
```

The program uses a function from the time module. The syntax used for the time module is actually the safer and more typical way to import a module. As you will see later in the program, the `sleep` function used from the `time` module will be referenced as `time.sleep()`. This tells the Python interpreter to look in the `time` module for the `sleep` function.

If we had used the `import` statement

```
from time import *
```

then the `sleep` function could just be referenced with `sleep()`. This is obviously easier, but it obscures the fact that the `sleep` function is not a part of the current module. Also several modules that a program imports might have functions with the same name. With the individual module name prefix, there is no ambiguity. Hence the form `import moduleName` is actually safer than `from moduleName import *`.

You might think that all modules could avoid using any of the same function names with a bit of planning. To get an idea of the magnitude of the issue, have a look at the number of modules available to Python. Try the following in the in the *Shell* (and likely wait a number of seconds):

```
help('modules')
```

Without module names to separate things out, it would be very hard to totally avoid name collisions with the enormous number of modules you see displayed, that are all available to Python!

Back to the current example program: The main program starts with standard window creation, and then makes three objects:

```
rect = Rectangle(Point(200, 90), Point(220, 100))
rect.setFill("blue")
rect.draw(win)

cir1 = Circle(Point(40,100), 25)
cir1.setFill("yellow")
cir1.draw(win)

cir2 = Circle(Point(150,125), 25)
cir2.setFill("red")
cir2.draw(win)
```

Zelle's reference pages do not mention the fact that the *order* in which these object are first *drawn* is significant. If objects overlap, the ones which used the draw method later appear on top. Other object methods like `setFill` or `move` do not alter which are in front of which. This becomes significant when `cir1` moves. The moving `cir1` goes over the rectangle and behind `cir2`. (Run the program again if you missed that.)

The animation starts with the code for a simple repeat loop:

```
for i in range(46):  # animate cir1 to the right
    cir1.move(5, 0)
    time.sleep(.05)
```

This very simple loop animates `cir1` moving in a straight line to the right. As in a movie, the illusion of continuous motion is given by jumping only a short distance each time (increasing the horizontal coordinate by 5). The `time.sleep` function, mentioned earlier, takes as parameter a time in seconds to have the program sleep, or delay, before continuing with the iteration of the loop. This delay is important, because modern computers are so fast, that the intermediate motion would be invisible without the delay. The delay can be given as a decimal, to allow the time to be a fraction of a second.

The next three lines are almost identical to the previous lines, and move the circle to the left (-5 in the horizontal coordinate each time).

```
for i in range(46):  # animate cir1 to the left
    cir1.move(-5, 0)
    time.sleep(.05)
```

The window closing lines of this program include a slight shortcut from earlier versions.

```
Text(Point(winWidth/2, 20), 'Click anywhere to quit.').draw(win)
```

The text object used to display the final message only needs to be referred to once, so a variable name is not necessary: The result of the Text object returned by the constructor is immediately used to draw the object. If the program needed to refer to this object again, this approach would not work.

The next example program, backAndForth1.py, it just a slight variation, looking to the user just like the last version. Only the small changes are shown below. This version was written after noticing how similar the two animation loops are, suggesting an improvement to the program: Animating any object to move in a straight line is a logical abstraction well expressed via a function.

The loop in the initial version of the program contained a number of arbitrarily chosen constants, which make sense to turn into parameters. Also, the object to be animated does not need to be `cir1`, it can be any of the drawable objects in the graphics package. The name `shape` is used to make this a parameter:

```
def moveOnLine(shape, dx, dy, repetitions, delay):
    for i in range(repetitions):
        shape.move(dx, dy)
        time.sleep(delay)
```

Then in the main function the two similar animation loops are reduced to a line for each direction:

```
moveOnLine(cir1, 5, 0, 46, .05)
moveOnLine(cir1, -5, 0, 46, .05)
```

Make sure you see these two lines with function calls behave the same way as the two animation loops in the main program of the original version.

Run the next example version, backAndForth2.py. The changes are more substantial here, and the display of the whole program is followed by display and discussion of the individual changes:

```
'''Test animation of a group of objects making a face.
'''
from graphics import *
import time

def moveAll(shapeList, dx, dy):
    ''' Move all shapes in shapeList by (dx, dy).'''
    for shape in shapeList:
        shape.move(dx, dy)


def moveAllOnLine(shapeList, dx, dy, repetitions, delay):
    '''Animate the shapes in shapeList along a line.
    Move by (dx, dy) each time.
    Repeat the specified number of repetitions.
    Have the specified delay (in seconds) after each repeat.
    '''

    for i in range(repetitions):
        moveAll(shapeList, dx, dy)
        time.sleep(delay)

def main():
    winWidth = 300
    winHeight = 300
    win = GraphWin('Back and Forth', winWidth, winHeight)
    win.setCoords(0, 0, winWidth, winHeight) # make right side up coordinates!

    rect = Rectangle(Point(200, 90), Point(220, 100))
    rect.setFill("blue")
    rect.draw(win)

    head = Circle(Point(40,100), 25)
    head.setFill("yellow")
    head.draw(win)

    eye1 = Circle(Point(30, 105), 5)
    eye1.setFill('blue')
    eye1.draw(win)

    eye2 = Line(Point(45, 105), Point(55, 105))
    eye2.setWidth(3)
    eye2.draw(win)

    mouth = Oval(Point(30, 90), Point(50, 85))
    mouth.setFill("red")
    mouth.draw(win)

    faceList = [head, eye1, eye2, mouth]
```

```
        cir2 = Circle(Point(150,125), 25)
        cir2.setFill("red")
        cir2.draw(win)

        moveAllOnLine(faceList, 5, 0, 46, .05)
        moveAllOnLine(faceList, -5, 0, 46, .05)

        Text(Point(winWidth/2, 20), 'Click anywhere to quit.').draw(win)
        win.getMouse()
        win.close()

    main()
```

Read the following discussion of program parts.

Moving a single elementary shape is rather limiting. It is much more interesting to compose a more complicated combination, like the face from the earlier example `face.py`. To animate such a combination, you cannot use the old `moveOnLine` function, because we want all the parts to move *together*, not one eye all the way across the screen and then have the other eye catch up! A variation on `moveOnLine` is needed where all the parts move together. We need all the parts of the face to move one step, sleep, and all move again, ... This could all be coded in a single method, but there are really two ideas here:

(1) Moving a group of objects one step.
(2) Animating a number of moves for the group.

This suggests two functions. Another issue is how to handle a group of elementary graphics objects. The most basic combination of objects in Python is a `list`, so we assume a parameter `shapeList`, which is a `list` of elementary graphics objects. For the first function, `moveAll,` just move all the objects in the list one step. Since we assume a *list* of objects and we want to move *each*, this suggests a for-each loop:

```
def moveAll(shapeList, dx, dy):
    ''' Move all shapes in shapeList by (dx, dy).'''
    for shape in shapeList:
        shape.move(dx, dy)
```

Having this function, we can easily write the second function `moveAllOnLine`, with a simple change from the `moveOnLine` function, substituting the `moveAll` function for the line with the `move` method:

```
def moveAllOnLine(shapeList, dx, dy, repetitions, delay):
    '''Animate the shapes in shapeList along a line.
    Move by (dx, dy) each time.
    Repeat the specified number of repetitions.
    Have the specified delay (in seconds) after each repeat.
    '''

    for i in range(repetitions):
        moveAll(shapeList, dx, dy)
        time.sleep(delay)
```

The code in main to construct the face is the same as in the earlier example `face.py`. Once all the pieces are constructed and colored, they must be placed in a list, for use in `moveAllOnLine`:

```
faceList = [head, eye1, eye2, mouth]
```

Then, later, the animation uses the `faceList` to make the face go back and forth:

```
    moveAllOnLine(faceList, 5, 0, 46, .05)
    moveAllOnLine(faceList, -5, 0, 46, .05)
```

This version of the program has encapsulated and generalized the moving and animating by creating functions and adding parameters that can be substituted. Again, make sure you see how the functions communicate to make the whole program work. This is an important and non-trivial use of functions.

Run the example program `backAndForth3.py`.

The final version, `backAndForth3.py`, uses the observation that the code to make a face embodies one unified idea, suggesting encapsulation inside a function. Once you have encapsulated the code to make a face, we can make several faces! Then the problem with the original code for the face is that all the positions for the facial elements are hard-coded: The face can only be drawn in one position. The full listing of `backAndForth3.py` below includes a `makeFace` function with a parameter for the position of the center of the face.

Beneath the listing of the whole program is a discussion of the individual changes:

```
'''Test animation of a group of objects making a face.
Combine the face elements in a function, and use it twice.
Have an extra level of repetition in the animation.
'''

from graphics import *
import time

def moveAll(shapeList, dx, dy):
    ''' Move all shapes in shapeList by (dx, dy).'''
    for shape in shapeList:
        shape.move(dx, dy)

def moveAllOnLine(shapeList, dx, dy, repetitions, delay):
    '''Animate the shapes in shapeList along a line.
    Move by (dx, dy) each time.
    Repeat the specified number of repetitions.
    Have the specified delay (in seconds) after each repeat.
    '''

    for i in range(repetitions):
        moveAll(shapeList, dx, dy)
        time.sleep(delay)

def makeFace(center, win):   #NEW
    '''display face centered at center in window win.
    Return a list of the shapes in the face.
    '''

    head = Circle(center, 25)
    head.setFill("yellow")
    head.draw(win)

    eye1Center = center.clone() # face positions are relative to the center
    eye1Center.move(-10, 5)     # locate further points in relation to others
    eye1 = Circle(eye1Center, 5)
    eye1.setFill('blue')
    eye1.draw(win)

    eye2End1 = eye1Center.clone()
    eye2End1.move(15, 0)
    eye2End2 = eye2End1.clone()
    eye2End2.move(10, 0)
    eye2 = Line(eye2End1, eye2End2)
    eye2.setWidth(3)
    eye2.draw(win)
```

```
    mouthCorner1 = center.clone()
    mouthCorner1.move(-10, -10)
    mouthCorner2 = mouthCorner1.clone()
    mouthCorner2.move(20, -5)
    mouth = Oval(mouthCorner1, mouthCorner2)
    mouth.setFill("red")
    mouth.draw(win)

    return [head, eye1, eye2, mouth]

def main():
    winWidth = 300
    winHeight = 300
    win = GraphWin('Back and Forth', winWidth, winHeight)
    win.setCoords(0, 0, winWidth, winHeight) # make right side up coordinates!

    rect = Rectangle(Point(200, 90), Point(220, 100))
    rect.setFill("blue")
    rect.draw(win)

    faceList = makeFace(Point(40, 100), win)    #NEW
    faceList2 = makeFace(Point(150,125), win)   #NEW

    stepsAcross = 46    #NEW section
    dx = 5
    dy = 3
    wait = .05
    for i in range(3):
        moveAllOnLine(faceList, dx, 0, stepsAcross, wait)
        moveAllOnLine(faceList, -dx, dy, stepsAcross//2, wait)
        moveAllOnLine(faceList, -dx, -dy, stepsAcross//2, wait)

    Text(Point(winWidth/2, 20), 'Click anywhere to quit.').draw(win)
    win.getMouse()
    win.close()

main()
```

Read the following discussion of program parts.

As mentioned above, the face construction function allows a parameter to specify where the center of the face is. The other parameter is the GraphWin that will contain the face.

```
def makeFace(center, win):
```

then the head is easily drawn, using this `center`, rather than `cir1` with specific center point (40, 100):

```
    head = Circle(center, 25)
    head.setFill("yellow")
    head.draw(win)
```

For the remaining `Points` used in the construction there is the issue of keeping the right relation to the center. This is accomplished much as in the creation of the second corner point in the `makeRectange` function in Section 2.4.6. A clone of the original center Point is made, and then moved by the *difference* in the positions of the originally specified `Points`. For instance, in the original face, the center of the head and first eye were at (40, 110) and (30, 115). That means a shift between the two coordinates of (-10, 5), since 30-40 = -10 and 130-110 = 20.

```
    eye1Center = center.clone() # face positions are relative to the center
        eye1Center.move(-10, 5)     # locate further points in relation to others
        eye1 = Circle(eye1Center, 5)
        eye1.setFill('blue')
        eye1.draw(win)
```

The only other changes to the face are similar, cloning and moving `Point`s, rather than specifying them with explicit coordinates.

```
        eye2End1 = eye1Center.clone()
        eye2End1.move(15, 0)
        eye2End2 = eye2End1.clone()
        eye2End2.move(10, 0)
        eye2 = Line(eye2End1, eye2End2)
        eye2.setWidth(3)
        eye2.draw(win)

        mouthCorner1 = center.clone()
        mouthCorner1.move(-10, -10)
        mouthCorner2 = mouthCorner1.clone()
        mouthCorner2.move(20, -5)
        mouth = Oval(mouthCorner1, mouthCorner2)
        mouth.setFill("red")
        mouth.draw(win)
```

Finally, the list of elements for the face must be returned to the caller:

```
    return [head, eye1, eye2, mouth]
```

Then in the main function, the program creates a face in exactly the same place as before, but using the `makeFace` function, with the original center of the face `Point(40, 100)`. Now with the `makeFace` function, with its center parameter, it is also easy to replace the old `cir2` with a whole face!

```
        faceList = makeFace(Point(40, 100), win)
        faceList2 = makeFace(Point(150,125), win)
```

The animation section is considerably elaborated in this version.

```
        stepsAcross = 46
        dx = 5
        dy = 3
        wait = .01
        for i in range(3):
            moveAllOnLine(faceList, dx, 0, stepsAcross, wait)
            moveAllOnLine(faceList, -dx, dy, stepsAcross//2, wait)
            moveAllOnLine(faceList, -dx, -dy, stepsAcross//2, wait)
```

The unidentified numeric literals that were used before are replaced by named values that easily identify the meaning of each one. This also allows the numerical values to be stated only once, allowing easy modification.

The whole animation is repeated three times by the use of a simple repeat loop.

The animations in the loop body illustrate that the straight line of motion does not need to be horizontal. The second and third lines use a non-zero value of both dx and dy for the steps, and move diagonally.

Make sure you see now how the whole program works together, including all the parameters for the moves in the loop.

By the way, the documentation of the functions in a module you have just run in the Shell is directly available. Try in the *Shell*:

```
    help(moveAll)
```

EXERCISE 2.4.8.1. ** Save backAndForth3.py to the new name backAndForth4.py. Add a triangular nose in the middle of the face in the `makeFace` function. Like the other features of the face, make sure the

position of the nose is *relative* to the center parameter. Make sure the nose is included in the final list of elements of the face that get returned.

Exercise 2.4.8.2. ** Make a program `faces.py` that asks the user to click the mouse, and then draws a face at the point where the user clicked. Elaborate this with a *simple repeat loop*, so a face appears for each of 6 clicks.

Exercise 2.4.8.3. ** Animate two faces moving in different directions *at the same time* in a program `move2Faces.py`. You cannot use the `moveAllOnLine` function. You will have to make a variation of your own. You *can* use the `moveAll` function separately for each face. Hint: imagine the old way of making an animated cartoon. If each face was on a separate piece of paper, and you wanted to animate them moving together, you would place them separately, record one frame, move them each a bit toward each other, record another frame, move each another bit toward each other, record another frame, ... In our animations "record a frame" is replaced by a short sleep to make the position visible to the user. Make a loop to incorporate the repetition of the moves.

2.4.9. Entry Objects. Read this section if you want to allow the user to enter text directly into a graphics window.

When using a graphics window, the shell window is still available. Keyboard input can be done in the normal text fashion, waiting for a response, and going on after the user presses the ENTER key. It is annoying to make a user pay attention to two windows, so the graphics module provides a way to enter text inside a graphics window, with the `Entry` type. The entry is a *partial* replacement for the `input` function.

Run the simple example, greet.py, which is copied below:

```
"""Simple example with Entry objects.
Enter your name, click the mouse, and see greetings.
"""

from graphics import *

def main():
    winWidth = 300
    winHeight = 300
    infoHeight = 15
    win = GraphWin("Greeting", winWidth, winHeight)
    win.setCoords(0,0, winWidth, winHeight)

    instructions = Text(Point(winWidth/2, 40),
                     "Enter your name.\nThen click the mouse.")
    instructions.draw(win)

    entry1 = Entry(Point(winWidth/2, 200),10)
    entry1.draw(win)
    Text(Point(winWidth/2, 230),'Name:').draw(win) # label for the Entry

    win.getMouse()  # To know the user is finished with the text.
    name = entry1.getText()

    greeting1 = 'Hello, ' + name + '!'
    Text(Point(winWidth/3, 150), greeting1).draw(win)

    greeting2 = 'Bonjour, ' + name + '!'
    Text(Point(2*winWidth/3, 100), greeting2).draw(win)

    instructions.setText("Click anywhere to quit.")
    win.getMouse()
    win.close()
```

```
main()
```

The only part of this with new ideas is:

```
entry1 = Entry(Point(winWidth/2, 200),10)
entry1.draw(win)
Text(Point(winWidth/2, 230),'Name:').draw(win) # label for the Entry

win.getMouse()  # To know the user is finished with the text.
name = entry1.getText()
```

The first line of this excerpt creates an Entry object, supplying its center point and a number of characters to leave space for (10 in this case).

As with other places where input is requested, a separate static label is added.

The way the underlying events are hidden in graphics.py, there is no signal when the user is done entering text in an `Entry` box. To signal the program, a mouse press is used above. In this case the location of the mouse press is not relevant, but once the mouse press is processed, execution can go on to reading the Entry text. The method name getText is the same as that used with a `Text` object.

Run the next example, addEntries.py, also copied below:

```
"""Example with two Entry objects and type conversion.
Do addition.
"""

from graphics import *

def main():
    winWidth = 300
    winHeight = 300

    win = GraphWin("Addition", winWidth, winHeight)
    win.setCoords(0,0, winWidth, winHeight)
    instructions = Text(Point(winWidth/2, 30),
                     "Enter two numbers.\nThen click the mouse.")
    instructions.draw(win)

    entry1 = Entry(Point(winWidth/2, 250),25)
    entry1.setText('0')
    entry1.draw(win)
    Text(Point(winWidth/2, 280),'First Number:').draw(win)

    entry2 = Entry(Point(winWidth/2, 180),25)
    entry2.setText('0')
    entry2.draw(win)
    Text(Point(winWidth/2, 210),'Second Number:').draw(win)

    win.getMouse()  # To know the user is finished with the text.

    numStr1 = entry1.getText()
    num1 = int(numStr1)
    numStr2 = entry2.getText()
    num2 = int(numStr2)
    result = "The sum of\n{num1}\nplus\n{num2}\nis {sum}.".format(**locals())
    Text(Point(winWidth/2, 110), result).draw(win)

    instructions.setText("Click anywhere to quit.")
```

```
    win.getMouse()
    win.close()

main()
```

There is not a separate graphical replacement for the `input` statement, so you only can read strings. With conversions, it is still possible to work with numbers.

Only one new graphical method has been included above:

```
    entry1.setText('0')
```

Again the same method name is used as with a `Text` object. In this case I chose not to leave the Entry initially blank. The 0 value also reinforces that a numerical value is expected.

There is also an `entry2` with almost identical code. After waiting for a mouse click, both entries are read, and the chosen names emphasizes they are strings. The strings must be converted to integers in order to do arithmetic and display the result.

The almost identical code for the two entries is a strong suggestion that this code could be written more easily with a function. You may look at the identically functioning example program `addEntries2.py`. The only changes are shown below. First there is a function to create an Entry and a centered static label over it.

```
def makeLabeledEntry(entryCenterPt, entryWidth, initialStr, labelText, win):
    '''Return an Entry object with specified center, width in characters, and
    initial string value.  Also create a static label over it with
    specified text.  Draw everything in the GraphWin win.
    '''

    entry = Entry(entryCenterPt, entryWidth)
    entry.setText(initialStr)
    entry.draw(win)
    labelCenter = entryCenterPt.clone()
    labelCenter.move(0, 30)
    Text(labelCenter,labelText).draw(win)
    return entry
```

In case I want to make more Entries with labels later, and refer to this code again, I put some extra effort in, making things be parameters even if only one value is used in this program. The position of the label is made 30 units above the entry by using the clone and move methods. Only the Entry is returned, on the assumption that the label is static, and once it is drawn, I can forget about it.

Then the corresponding change in the main function is just two calls to this function:

```
    entry1 = makeLabeledEntry(Point(winWidth/2, 250), 25,
                              '0', 'First Number:', win)
    entry2 = makeLabeledEntry(Point(winWidth/2, 180), 25,
                              '0', 'Second Number:', win)
```

These lines illustrate that a statement may take more than one line. In particular, as in the Shell, Python is smart enough to realize that there must be a continuation line if the parentheses do not match.

While I was improving things, I also changed the conversions to integers. In the first version I wanted to emphasize the existence of both the string and integer data as a teaching point, but the num1Str and num2Str variables were only used once, so a more concise way to read and convert the values is to eliminate them:

```
    num1 = int(entry1.getText())
    num2 = int(entry2.getText())
```

2.4.10. Color Names. Thus far we have only used common color names. In fact there are a very large number of allowed color names, and the abiltity to draw with custom colors.

First, the graphics package is built on an underlying graphics system, Tkinter, which has a large number of color names defined. Each of the names can be used by itself, like 'red', 'salmon' or 'aquamarine' or with a lower intensity by specifying with a trailing number 2, 3, or 4, like 'red4' for a dark red.

Though the ideas for the coding have not all been introduced, it is still informative to *run* the example program colors.py. As you click the mouse over and over, you see the names and appearances of a wide variety of built-in color names. The names must be place in quotes, but capitalization is ignored.

2.4.11. Custom Colors. Custom colors can also be created. To do that requires some understanding of human eyes and color (and the Python tools). The only colors detected directly by the human eyes are red, green, and blue. Each amount is registered by a different kind of cone cell in the retina. As far as the eye is concerned, all the other colors we see are just combinations of these three colors. This fact is used in color video screens: they only directly display these three colors. A common scale to use in labeling the intensity of each of the basic colors (red, green, blue) is from 0 to 255, with 0 meaning none of the color, and 255 being the most intense. Hence a color can be described by a sequence of red, green, and blue intensities (often abbreviated RGB). The graphics package has a function, `color_rgb`, to create colors this way. For instance a color with about half the maximum red intensity, no green, and maximum blue intensity would be

```
aColor = color_rgb(128, 0, 255)
```

Such a creation can be used any place a color is used in the graphics, (i.e. `circle.setFill(aColor)`).

2.4.12. Random Colors. Another interesting use of the `color_rgb` function is to create random colors. Run example program `randomCircles.py`. The code also is here:

```
"""Draw random circles.
"""

from graphics import *
import random, time

def main():
    win = GraphWin("Random Circles", 300, 300)
    for i in range(75):
        r = random.randrange(256)
        b = random.randrange(256)
        g = random.randrange(256)
        color = color_rgb(r, g, b)

        radius = random.randrange(3, 40)
        x = random.randrange(5, 295)
        y = random.randrange(5, 295)

        circle = Circle(Point(x,y), radius)
        circle.setFill(color)
        circle.draw(win)
        time.sleep(.05)

    Text(Point(150, 20), "Click to close.").draw(win)
    win.getMouse()
    win.close()

main()
```

Read the fragments of this program and their explanations:

To do random things, the program needs a function from the random module. This example shows that imported modules may be put in a comma separated list:

```
import random, time
```

You have already seen the built-in function `range`. To generate a sequence of all the integers 0, 1, ... 255, you would use

```
range(256)
```

This is the full list of possible values for the red, green or blue intensity parameter. For this program we randomly choose any *one* element from this sequence. Instead of the `range` function, use the `random` module's `randrange` function, as in

```
r = random.randrange(256)
b = random.randrange(256)
g = random.randrange(256)
color = color_rgb(r, g, b)
```

This gives randomly selected values to each of `r`, `g`, and `b`, which are then used to create the random `color`.

I want a random circle radius, but I do not want a number as small as 0, making it invisible. The `range` and `randrange` functions both refer to a possible sequence of values starting with 0 when a *single* parameter is used. It is also possible to add a different starting value as the *first* parameter. You still must specify a value *past* the end of the sequence. For instance

```
range(3, 40)
```

would refer to the sequence `3, 4, 5, ... , 39` (starting with 3 and not quite reaching 40). Similarly

```
random.randrange(3, 40)
```

randomly selects an arbitrary element of `range(3, 40)`.

I use the two-parameter version to select random parameters for a Circle:

```
radius = random.randrange(3, 40)
x = random.randrange(5, 295)
y = random.randrange(5, 295)

circle = Circle(Point(x,y), radius)
```

What are the smallest and largest values I allow for x and y? [7]

Random values are often useful in games.

EXERCISE 2.4.12.1. * Write a program `ranges.py` that uses the range function to produce the sequnce 1, 2, 3, 4, and then print it. Also prompt the user for an integer n and print the sequnce 1, 2, 3, ... , n – *including* n. Hint: [8] Finally use a simple repeat loop to find and print five randomly chosen numbers from the range 1, 2, 3, ... , n .

2.5. Files

This section fits here logically (as an important built-in type of object) but it is not needed for the next chapter, Flow of Control, 3.

Thus far you have been able to save programs, but anything produced during the execution of a program has been lost when the program ends. Data has not *persisted* past the end of execution. Just as programs live on in files, you can generate and read data files in Python that persist after your program has finished running.

As far as Python is concerned, a file is just a string (often very large!) stored on your file system, that you can read or write gradually or all together.

Open a directory window for your Python program directory. First note that there is no file named sample.txt.

Make sure you have started Idle so the current directory is your Python program directory (for instance *in Windows with the downloaded shortcut to Idle*).

Run the example program `firstFile.py`, shown below:

```
outFile = open('sample.txt', 'w')
outFile.write('My first output file!')
outFile.close()
```

[7] 5 and 294 (one less than 295).

[8] If 4 or n is the last number, what is the first number *past* the end of the sequence?

The first line creates a file object, which links Python to your computer's file system. The first parameter in the file constructor gives the file name, `sample.txt`. The second parameter indicates how you use the file. The `'w'` is short for *w*rite, so you will be creating and *writing* to a file (or if it already existed, destroying the old contents and starting over!). If you do not use any operating system directory separators in the name ('\' or '/' depending on your operating system), then the file will lie in the current directory. The assignment statement gives the python file object the name `outFile`.

The second line writes the specified string to the file.

The last line is important to clean up. Until this line, this Python program controls the file, and nothing may even be actually written to the file. (Since file operations are thousands of times slower than memory operations, Python *buffers* data, saving small amounts and writing all at once in larger chunks.) The `close` line is essential for Python to make sure everything is really written, and to relinquish control of the file. It is a common bug to write a program where you have the code to add all the data you want to a file, but the program does not end up creating a file. Usually this means you forgot to close the file.

Now switch focus and look at a file window for the current directory. You should now see a file `sample.txt`. You can open it in Idle (or your favorite word processor) and see its contents.

Run the example program `nextFile.py`, shown below, which has two calls to the write method:

```
outFile = open('sample2.txt', 'w')
outFile.write('My second output file!
outFile.write('Write some more.')
outFile.close()
```

Now look at the file, `sample2.txt`. Open it in Idle. It may not be what you expect! The write method for the file is not quite like a `print` function. It does not add anything to the file except *exactly* the data you tell it to write. If you want a newline, you must indicate it *explicitly*. Recall the newline code `\n`. Run the example program `revisedFile.py`, shown below, which adds newline codes:

```
outFile = open('sample3.txt', 'w')
outFile.write('A revised output file!\n')
outFile.write('Write some more.\n')
outFile.close()
```

Check the contents of sample3.txt. This manner of checking the file shows it is really in the file system, but the focus in the Tutorial should be on using Python! Run the example program `printFile.py`, shown below:

```
inFile = open('sample3.txt', 'r')
contents = inFile.read()
print(contents)
```

Now you have come full circle: what one Python program has written into the file sample3.txt, another has read and displayed.

In the first line an operating system file (`sample3.txt`) is associated again with a Python variable name (`inFile`). The second parameter again gives the mode of operation, but this time it is `'r'`, short for *r*ead. This file, `sample3.txt`, should already exist, and the intention is to read from it. This is the most common mode for a file, so the `'r'` parameter is actually *optional*.

The `read` method returns all the file's data as a single string, here assigned to `contents`. Using the `close` method is generally optional with files being read. There is nothing to lose if a program ends without closing a file that was being read.[9]

EXERCISE 2.5.0.2. Make the following programs in sequence. Be sure to save the programs in the same directory as where you start the idle shortcut and where you have all the sample text files:
* a. printUpper.py: read the contents of the sample2.txt file and print the contents out in upper case. (This should use file operations and should work no matter what the contents are in sample2.txt. Do not assume the particular string written by `nextFile.py`!)
* b. fileUpper.py: prompt the user for a file name, read and print the contents of the requested file in upper case.

[9]If, for some reason, you want to reread this same file while the same program is running, you need to close it and reopen it.

** c. copyFileUpper: modify fileUpper.py to write the upper case string to a new file rather than printing it. Have the name of the new file be dynamically derived from the old name by prepending 'UPPER' to the name. For example, if the user specified the file `sample.txt` (from above), the program would create a file `UPPERsample.txt`, containing 'MY FIRST OUTPUT FILE!'. When the user specifies the file name `stuff.txt`, the resulting file would be named `UPPERstuff.txt`.

EXERCISE 2.5.0.3. Write `madlib3.py`, a small modification of `madlib2.py`, requiring only a modification to the main function of `madlib2.py`. Your `madlib3.py` should prompt the user for the name of a file that should contain a madlib format string as text (with no quotes around it). Read in this file and use it as the format string in the `tellStory` function. This is unlike in `madlib2.py`, where the story is a literal string coded directly into the program called originalStory. The `tellstory` function and particularly the `getKeys` function were developed and described in detail in this tutorial, but for this exercise there is no need to follow their inner workings - you are just a user of the `tellstory` function (and the functions that it calls). You do not need to mess with the code for the definition of `tellStory` or any of the earlier supporting functions. The original madlib string is already placed in a file `jungle.txt`, that is in this format as an example. With the Idle editor, write another madlib format string into a file `myMadlib.txt.` If you earlier created a file `myMadlib.py`, then you can easily extract the story from there (without the quotes around it). Test your program both with `jungle.txt` and your new madlib story file.

2.6. Summary

The same typographical conventions will be used as in the last summary in Section 1.15.

(1) Object notation
  - (a) When the name of a type of object is used as a function call, it is called a *constructor*, and a new object of that type is constructed and returned. The meanings of any parameters to the constructor depend on the type. [2.2.3]
  - (b) `object.methodName(parameters)`
    Objects may have special operations associated with them, called methods. They are functions automatically applied to the *object* before the dot. Further parameters may be expected, depending on the particular method. [2.1.1]

(2) String (`str`) indexing and methods
  See the Chapter 1 Summary Section 1.15 for string literals and symbolic string operations.
  - (a) String Indexing. [2.1.2]
    stringReference[intExpression]
    Individual characters in a string may be chosen. If the string has length L, then the indices start from 0 for the initial character and go to L-1 for the rightmost character. Negative indices may also be used to count from the right end, -1 for the rightmost character through -L for the leftmost character. Strings are immutable, so individual characters may be read, but not set.
  - (b) String Slices [2.1.3]
    stringReference[start : pastEnd]
    stringReference[start :]
    stringReference[: pastEnd]
    stringReference[:]
    A substring or *slice* of 0 or more consecutive characters of a string may be referred to by specifying a starting index and the index one *past* the last character of the substring. If the starting or ending index is left out Python uses 0 and the length of the string respectively. Python assumes indices that would be beyond an end of the string actually mean the end of the string.
  - (c) String Methods: Assume *s* refers to a string
    - (i) `s.upper()`
      Returns an uppercase version of the string *s*. [2.1.1]
    - (ii) `s.lower()`
      Returns a lowercase version of the string *s*.[2.1.1]

(iii) *s*.count(*sub*)
Returns the number of repetitions of the substring *sub* inside *s*. [2.1.1]

(iv) *s*.find(*sub*)
s.find(*sub*, *start*)
s.find(*sub*, *start*, *end*)
Returns the index in *s* of the first character of the first occurrence of the substring *sub* within the part of the string *s* indicated, respectively the whole string *s*, s[*start* :], or *s*[*start* : *end*], where *start* and *end* have integer values.[2.1.1]

(v) *s*.split()
s.split(*sep*)
The first version splits *s* at any sequence of whitespace (blanks, newlines, tabs) and returns the remaining parts of *s* as a list. If a string *sep* is specified, it is the separator that gets removed from between the parts of the list. [2.1.5]

(vi) s*ep*.join(*sequence*)
Return a new string obtained by joining together the *sequence* of strings into one string, interleaving the string *sep* between *sequence* elements. [2.1.6]

(vii) Further string methods are discussed in the Python Reference Manual, Section 2.3.6.1, String Methods. [2.1.7]

(3) Sets
A `set` is a collection of elements with no repetitions. It can be used as a sequence in a `for` loop. A `set` constructor can take any other sequence as a parameter, and convert the sequence to a `set` (with no repetitions). Nonempty set literals are enclosed in braces. [2.2.2]

(4) List method `append`
aList.append(*element*)
Add an arbitrary *element* to the end of the `list` *aList*, *mutating* the list, not returning any list. [2.2.1]

(5) Files [2.5]
`file(`*nameInFileSystem*`)` returns a file object for reading, where *nameInFileSystem* must be a string referring to an existing file. The file location is relative to the current directory.
`file(`*nameInFileSystem*`, 'w')` returns a file object for writing, where the string *nameInFileSystem* will be the name of the file. If it did not exist before, it is created. CAUTION: If it *did* exist before, all previous contents are erased. The file location is relative to the current directory.
If `infile` is a file opened for reading, and `outfile` is a file opened for writing, then
`infile.read()` returns the entire file contents of the file as a string.
`infile.close()` closes the file in the operating system (generally not needed, unless the file is going to be modified later, while your program is *still* running).
`outfile.write(`*stringExpression*`)` writes the string to the file, with no extra newline.
`outfile.close()` closes the file in the operating system (*important* to make sure the whole file gets written and to allow other access to the file).

(6) Mutable objects [2.4.6]
Care must be taken whenever a second name is assigned to a mutable object. It is an *alias* for the original name, and refers to the exact same object. A mutating method applied to either name changes the one object referred to by both names.

Many types of mutable object have ways to make a copy that is a distinct object. Zelle's graphical objects have the `clone` method. A copy of a list may be made with a full slice: *someList*[:]. Then direct mutations to one list (like appending an element) do not affect the other list, but still, each list is indirectly changed if a common mutable element in the lists is changed.

(7) Graphics
A systematic reference to Zelle's graphics package, graphics.py, is at `http://mcsp.wartburg.edu/zelle/python/graphics/graphics/index.html`.

(a) Introductory examples of using graphics.py are in [2.4.1], [2.4.2], and [2.4.9]
(b) Windows operating system .pyw
In windows, a graphical program that take no console input and generates no console output, may be given the extension .pyw to suppress the generation of a console window. [2.4.3]
(c) Event-driven programs
Graphical programs are typically event-driven, meaning the next operation done by the program can be in response to a large number of possible operations, from the keyboard or mouse for instance, without the program knowing which kind of event will come next. For simplicity, this approach is pretty well hidden under Zelle's graphics package, allowing the illusion of simpler sequential programming. [2.4.4]
(d) Custom computer colors are expressed in terms of the amounts of red, green, and blue. [2.4.11]
(e) See also Animation under the summary of Programming Techniques.

(8) Additional programming techniques
(a) These techniques extend those listed in the summary of the previous chapter. [1.15]
(b) Sophisticated operations with substrings require careful setting of variables used as an index. [2.1.4]
(c) There are a number of techniques to assist creative programming, including pseudo-code and gradual generalization from concrete examples. [2.3.2]
(d) Animation: a loop involving small moves followed by a short delay (assumes the time module is imported): [2.4.8

```
loop heading :
    move all objects a small step in the proper direction
    time.sleep(delay).
```

(e) Example of a practical successive modification loop: [2.3.1]
(f) Examples of encapsulating ideas in functions and reusing them: [2.3.1], [2.3.3], [2.4.8]
(g) Random results can be introduced into a program using the random module. [2.4.12]

CHAPTER 3

More On Flow of Control

You have varied the normal forward sequence of operations with functions and `for` loops. To have full power over your programs, you need two more constructions that changing the flow of control: decisions choosing between alternatives (`if` statements), and more general loops that are not required to be controlled by the elements of a collection (`while` loops).

3.1. If Statements

3.1.1. Simple Conditions. The statements introduced in this chapter will involve tests or conditions. More syntax for conditions will be introduced later, but for now consider simple arithmetic comparisons that directly translate from math into Python. Try each line separately in the *Shell*

```
2 < 5
3 > 7
x = 11
x > 10
2*x < x
type(True)
```

You see that conditions are either `True` or `False` (with no quotes). These are the only possible *Boolean* values (named after 19th century mathematician George Boole). In Python the name Boolean is shortened to the type `bool`. It is the type of the results of true-false tests.

3.1.2. Simple `if` Statements. Run this example program, suitcase.py. Try it at least twice, with inputs: 30 and then 55. As you an see, you get an extra result, depending on the input. The main code is:

```
weight = float(input('How many pounds does you suitcase weigh? '))
if weight > 50:
    print('There is a $25 charge for luggage that heavy.')
print('Thank you for your business.')
```

The middle two line are an `if`-statement. It reads pretty much like English. If it is true that the weight is greater than 50, then print the statement about an extra charge. If it is not true that the weight is greater than 50, then don't do the indented part: skip printing the extra luggage charge. In any event, when you have finished with the if-statement (whether it actually does anything or not), go on to the next statement that is *not* indented under the `if`. In this case that is the statement printing "Thank you".

The general Python syntax for a simple `if` statement is

```
if condition :
    indentedStatementBlock
```

If the condition is true, then do the indented statements. If the condition is not true, then skip the indented statements.

Another fragment as an example:

```
if balance < 0:
    transfer = -balance
    backupAccount = backupAccount - transfer # take enough from the backup acct.
    balance = balance + transfer
```

As with other kinds of statements with a heading and an indented block, the block can have more than one statement. The assumption in the example above is that if an account goes negative, it is brought back to 0 by transferring money from a backup account in several steps.

In the examples above the choice is between doing something (if the condition is `True`) or nothing (if the condition is `False`). Often there is a choice of two possibilities, only one of which will be done, depending on the truth of a condition.

3.1.3. `if-else` Statements. Run the example program, clothes.py. Try it at least twice, with inputs: 50, 80. As you can see, you get different results, depending on the input. The main code of `clothes.py` is:

```
temperature = float(input('What is the temperature? '))
if temperature > 70:
    print('Wear shorts.')
else:
    print('Wear long pants.')
print('Get some exercise outside.')
```

The middle four lines are an `if-else` statement. Again it is close to English, though you might say "otherwise" instead of "else" (but else is shorter!). There are two indented blocks: One, like in the simple `if` statement, comes right after the `if` heading and is executed when the condition in the `if` heading is true. In the `if-else` form this is followed by an `else:` line, followed by another indented block that is only executed when the original condition is *false*. In an if-else statement exactly one of two possible indented blocks is executed.

A line is also shown *out*dented next, about getting exercise. Since it is outdented, it is not a part of the if-else statement: It is always executed in the normal forward flow of statements, after the `if-else` statement (whichever block is selected).

The general Python syntax is

```
if condition :
    indentedStatementBlockForTrueCondition
else:
    indentedStatementBlockForFalseCondition
```

These statement blocks can have any number of statements, and can include about any kind of statement.

3.1.4. More Conditional Expressions. All the usual arithmetic comparisons may be made, but many do not use standard mathematical symbolism, mostly for lack of proper keys on a standard keyboard.

Meaning	Math Symbol	Python Symbols
Less than	$<$	`<`
Greater than	$>$	`>`
Less than or equal	$\leq$	`<=`
Greater than or equal	$\geq$	`>=`
Equals	$=$	`==`
Not equal	$\neq$	`!=`

There should not be space between the two-symbol Python substitutes.

Notice that the obvious choice for *equals*, a single equal sign, is *not* used to check for equality. An annoying second equal sign is required. This is because the single equal sign is already used for *assignment* in Python, so it is not available for tests. It is a common error to use only one equal sign when you mean to *test* for equality, and not make an assignment!

Tests for equality do not make an assignment, and they do not require a variable on the left. Any expressions can be tested for equality or inequality (!=). They do not need to be numbers! Predict the results and try each line in the *Shell*:

```
x = 5
x
x == 5
x == 6
x
x != 6
x = 6
```

```
6 == x
6 != x
'hi' == 'h' + 'i'
'HI' != 'hi'
[1, 2] != [2, 1]
```

An equality check does not make an assignment. Strings are case sensitive. Order matters in a list.

Try in the *Shell*:

```
'a' > 5
```

When the comparison does not make sense, an Exception is caused.[1]

EXERCISE 3.1.4.1. Write a program, `graduate.py`, that prompts students for how many credits they have. Print whether of not they have enough credits for graduation. (At Loyola University Chicago 128 credits are needed for graduation.)

EXERCISE 3.1.4.2. Following up on the discussion of the *inexactness* of float arithmetic in Section 1.14.3, confirm that Python does not consider .1 + .2 to be equal to .3: Write a simple condition into the Shell to test.

Here is another example: Pay with Overtime. Given a person's work hours for the week and regular hourly wage, calculate the total pay for the week, taking into account overtime. Hours worked over 40 are overtime, paid at 1.5 times the normal rate. This is a natural place for a function enclosing the calculation.

Read the setup for the function:

```
def calcWeeklyWages(totalHours, hourlyWage):
    '''Return the total weekly wages for a worker working totalHours,
    with a given regular hourlyWage.  Include overtime for hours over 40.
    '''
```

The problem clearly indicates two cases: when no more than forty hours are worked or when more than 40 hours are worked. In case more than 40 hours are worked, it is convenient to introduce a variable overtimeHours. You are encouraged to think about a solution before going on and examining mine.

You can try running my complete example program, wages.py, also shown below. The format operation at the end of the main function uses the floating point format (Section 1.14.3) to show two decimal places for the cents in the answer:

```
def calcWeeklyWages(totalHours, hourlyWage):
    '''Return the total weekly wages for a worker working totalHours,
    with a given regular hourlyWage.  Include overtime for hours over 40.
    '''

    if totalHours <= 40:
        totalWages = hourlyWage*totalHours
    else:
        overtime = totalHours - 40
        totalWages = hourlyWage*40 + (1.5*hourlyWage)*overtime
    return totalWages

def main():
    hours = float(input('Enter hours worked: '))
    wage = float(input('Enter dollars paid per hour: '))
    total = calcWeeklyWages(hours, wage)
    print('Wages are ${total:.2f}.'.format(**locals()))

main()
```

Here the input was intended to be numeric, but it could be decimal so the conversion from string was via `float`, not `int`.

[1]This is an improvement that is new in Python 3.0

Below is an equivalent alternative version of the body of `calcWeeklyWages`, used in `wages1.py`. It uses just one general calculation formula and sets the parameters for the formula in the if statement. There are generally a number of ways you might solve the same problem!

```
if totalHours <= 40:
    regularHours = totalHours
    overtime = 0
else:
    overtime = totalHours - 40
    regularHours = 40
return hourlyWage*regularHours + (1.5*hourlyWage)*overtime
```

3.1.5. Multiple Tests and `if-elif` Statements . Often you want to distinguish between more than two distinct cases, but conditions only have two possible results, `True` or False, so the only direct choice is between two options. As anyone who has played "20 Questions" knows, you can distinguish more cases by further questions. If there are more than two choices, a single test may only reduce the possibilities, but further tests can reduce the possibilities further and further. Since most any kind of statement can be placed in an indented statement block, one choice is a further `if` statement. For instance consider a function to convert a numerical grade to a letter grade, 'A', 'B', 'C', 'D' or 'F', where the cutoffs for 'A', 'B', 'C', and 'D' are 90, 80, 70, and 60 respectively. One way to write the function would be test for one grade at a time, and resolve all the remaining possibilities inside the next `else` clause:

```
def letterGrade(score):
    if score >= 90:
        letter = 'A'
    else:   # grade must be B, C, D or F
        if score >= 80:
            letter = 'B'
        else:  # grade must be C, D or F
            if score >= 70:
                letter = 'C'
            else:    # grade must D or F
                if score >= 60:
                    letter = 'D'
                else:
                    letter = 'F'
    return letter
```

This repeatedly increasing indentation with an `if` statement as the `else` block can be annoying and distracting. A preferred alternative in this situation, that avoids all this indentation, is to combine each `else` and `if` block into an `elif` block:

```
def letterGrade(score):
    if score >= 90:
        letter = 'A'
    elif score >= 80:
        letter = 'B'
    elif score >= 70:
        letter = 'C'
    elif score >= 60:
        letter = 'D'
    else:
        letter = 'F'
    return letter
```

The most elaborate syntax for an if statement, `if-elif-...-else` is indicated in general below:

```
if condition1 :
```

```
    indentedStatementBlockForTrueCondition1
elif condition2 :
    indentedStatementBlockForFirstTrueCondition2
elif condition3 :
    indentedStatementBlockForFirstTrueCondition3
elif condition4 :
    indentedStatementBlockForFirstTrueCondition4
else:
    indentedStatementBlockForEachConditionFalse
```

The `if`, each `elif`, and the final `else` line are all aligned. There can be any number of `elif` lines, each followed by an indented block. (Three happen to be illustrated above.) With this construction *exactly one* of the indented blocks is executed. It is the one corresponding to the *first* `True` condition, or, if all conditions are `False`, it is the block after the final `else` line.

Be careful of the strange Python contraction. It is `elif`, not `elseif`. A program testing the letterGrade function is in example program `grade1.py`.

EXERCISE 3.1.5.1. In Idle, load `grade1.py` and save it as `grade2.py` Modify `grade2.py` so it has an equivalent version of the letterGrade function that tests in the opposite order, first for F, then D, C, ... Hint: How many tests do you need to do? [2] Be sure to run your new version and test with different inputs that test all the different paths through the program.

EXERCISE 3.1.5.2. Write a program `sign.py` to ask the user for a number. Print out which category the number is in: `'positive'`, `'negative'`, or `'zero'`.

EXERCISE 3.1.5.3. Modify the `wages.py` or the `wages1.py` example to create a program `wages2.py` that assumes people are paid double time for hours over 60. Hence they get paid for at most 20 hours overtime at 1.5 times the normal rate. For example, a person working 65 hours with a regular wage of $10 per hour would work at $10 per hour for 40 hours, at 1.5*$10 for 20 hours of overtime, and 2*$10 for 5 hours of double time, for a total of 10*40 + (1.5*10)*20 + (2*10)*5 = $800. You may find `wages1.py` easier to adapt than `wages.py`.

A final alternative for `if` statements: `if-elif-`.... with *no* `else`. This would mean changing the syntax for if-elif-else above so the final `else:` and the block after it would be omitted. It is similar to the basic `if` statement without an `else`, in that it is possible for *no* indented block to be executed. This happens if *none* of the conditions in the tests are true.

With an `else` included, *exactly* one of the indented blocks is executed. Without an `else`, at *most* one of the indented blocks is executed.

```
if weight > 120:
    print('Sorry, we can not take a suitcase that heavy.')
elif weight > 50:
    print('There is a $25 charge for luggage that heavy.')
```

This `if-elif` statement only prints a line if there is a problem with the weight of the suitcase.

3.1.6. Nesting Control-Flow Statements, Part I. The power of a language like Python comes largely from the variety of ways basic statements can be combined. In particular, `for` and `if` statements can be nested inside each other's indented blocks. For example, suppose you want to print only the positive numbers from an arbitrary list of numbers in a function with the following heading. *Read* the pieces for now.

```
def printAllPositive(numberList):
    '''Print only the positive numbers in numberList.'''
```

For example, suppose `numberList` is `[3, -5, 2, -1, 0, 7]`. As a human, who has eyes of amazing capacity, you are drawn immediately to the actual correct numbers, 3, 2, and 7, but clearly a computer doing this systematically will have to check *every* number. That easily suggests a for-each loop starting

```
for num in numberList:
```

[2] 4 tests to distinguish the 5 cases, as in the previous version

What happens in the body of the loop is not the same thing each time: *some* get printed, and for those we will want the statement

```
print(num)
```

but some do not get printed, so it may at first seem that this is not an appropriate situation for a for-*each* loop, but in fact, there is a *consistent* action required: Every number must be tested to see *if* it should be printed. This suggests an `if` statement, with the condition `num > 0`. Try loading into Idle and running the example program `onlyPositive.py`, whose code is shown below. It ends with a line testing the function:

```
def printAllPositive(numberList):
    '''Print only the positive numbers in numberList.'''
    for num in numberList:
        if num > 0:
            print(num)

printAllPositive([3, -5, 2, -1, 0, 7])
```

This idea of nesting if statements enormously expands the possibilities with loops. Now different things can be done at different times in loops, as long as there is a *consistent test* to allow a choice between the alternatives.

The rest of this section deals with graphical examples.

Run example program `bounce1.py`. It has a red ball moving and bouncing obliquely off the edges. If you watch several times, you should see that it starts from random locations. Also you can repeat the program from the Shell prompt after you have run the script. For instance, right after running the program, try in the *Shell*

```
bounceBall(-3, 1)
```

The parameters give the amount the shape moves in each animation step. You can try other values in the *Shell*, preferably with magnitudes less than 10.

For the remainder of the description of this example, *read* the extracted text pieces.

The animations before this were totally scripted, saying exactly how many moves in which direction, but in this case the direction of motion changes with every bounce. The program has a graphic object `shape` and the central animation step is

```
shape.move(dx, dy)
```

but in this case, dx and dy have to change when the ball gets to a boundary. For instance, imagine the ball getting to the left side as it is moving to the left and up. The bounce obviously alters the horizontal part of the motion, in fact reversing it, but the ball would still continue up. The reversal of the horizontal part of the motion means that the horizontal shift changes direction and therefore its sign:

```
dx = -dx
```

but `dy` does not need to change. This switch does not happen at each animation step, but only when the ball reaches the edge of the window. It happens only some of the time – suggesting an `if` statement. Still the condition must be determined. Suppose the center of the ball has coordinates (x, y). When x reaches some particular x coordinate, call it xLow, the ball should bounce.

The edge of the window is at coordinate 0, but `xLow` should not be 0, or the ball would be half way off the screen before bouncing! For the edge of the ball to hit the edge of the screen, the x coordinate of the center must be the length of the radius away, so actually `xLow` is the radius of the ball.

Animation goes quickly in small steps, so I cheat. I allow the ball to take one (small, quick) step past where it really should go (`xLow`), and then we reverse it so it comes back to where it belongs. In particular

```
if x < xLow:
    dx = -dx
```

There are similar bounding variables `xHigh`, `yLow` and `yHigh`, all the radius away from the actual edge coordinates, and similar conditions to test for a bounce off each possible edge. Note that whichever edge is hit, one coordinate, either dx or dy, reverses. One way the collection of tests could be written is

```
if x < xLow:
    dx = -dx
if x > xHigh
```

```
    dx = -dx
if y < yLow:
    dy = -dy
if y > yHigh
    dy = -dy
```

This approach would cause there to be some extra testing: If it is true that `x < xLow`, then it is impossible for it to be true that `x > xHigh`, so we do not need both tests together. We avoid unnecessary tests with an elif clause (for both x and y):

```
if x < xLow:
    dx = -dx
elif x > xHigh
    dx = -dx
if y < yLow:
    dy = -dy
elif y > yHigh
    dy = -dy
```

Note that the middle `if` is *not* changed to an `elif`, because it is possible for the ball to reach a *corner*, and need both `dx` and `dy` reversed.

The program also uses several accessor methods for graphics objects that we have not used in examples yet. Various graphics objects, like the circle we are using as the shape, know their center point, and it can be accessed with the `getCenter()` method. (Actually a clone of the point is returned.) Also each coordinate of a `Point` can be accessed with the `getX()` and `getY()` methods.

This explains the new features in the central function defined for bouncing around in a box, `bounceInBox`. The animation arbitrarily goes on in a simple repeat loop for 600 steps. (A later example will improve this behavior.):

```
def bounceInBox(shape, dx, dy, xLow, xHigh, yLow, yHigh):
    ''' Animate a shape moving in jumps (dx, dy), bouncing when
    its center reaches the low and high x and y coordinates.
    '''

    delay = .005
    for i in range(600):
        shape.move(dx, dy)
        center = shape.getCenter()
        x = center.getX()
        y = center.getY()
        if x < xLow:
            dx = -dx
        elif x > xHigh:
            dx = -dx

        if y < yLow:
            dy = -dy
        elif y > yHigh:
            dy = -dy
        time.sleep(delay)
```

The program starts the ball from an arbitrary point inside the allowable rectangular bounds. This is encapsulated in a utility function included in the program, `getRandomPoint`. The getRandomPoint function uses the `randrange` function from the module `random`. Note that in parameters for both the functions `range` and `randrange`, the end stated is *past* the last value actually desired:

```
def getRandomPoint(xLow, xHigh, yLow, yHigh):
    '''Return a random Point with coordinates in the range specified.'''
    x = random.randrange(xLow, xHigh+1)
```

```
    y = random.randrange(yLow, yHigh+1)
    return Point(x, y)
```

The full program is listed below, repeating `bounceInBox` and `getRandomPoint` for completeness. Several parts that may be useful later, or are easiest to follow as a unit, are separated out as functions. Make sure you see how it all hangs together or ask questions!

```
'''
Show a ball bouncing off the sides of the window.
'''
from graphics import *
import time, random

def bounceInBox(shape, dx, dy, xLow, xHigh, yLow, yHigh):
    ''' Animate a shape moving in jumps (dx, dy), bouncing when
    its center reaches the low and high x and y coordinates.
    '''

    delay = .005
    for i in range(600):
        shape.move(dx, dy)
        center = shape.getCenter()
        x = center.getX()
        y = center.getY()
        if x < xLow:
            dx = -dx
        elif x > xHigh:
            dx = -dx

        if y < yLow:
            dy = -dy
        elif y > yHigh:
            dy = -dy
        time.sleep(delay)

def getRandomPoint(xLow, xHigh, yLow, yHigh):
    '''Return a random Point with coordinates in the range specified.'''
    x = random.randrange(xLow, xHigh+1)
    y = random.randrange(yLow, yHigh+1)
    return Point(x, y)

def makeDisk(center, radius, win):
    '''return a red disk that is drawn in win with given center and radius.'''

    disk = Circle(center, radius)
    disk.setOutline("red")
    disk.setFill("red")
    disk.draw(win)
    return disk

def bounceBall(dx, dy):
    '''Make a ball bounce around the screen, initially moving by (dx, dy)
    at each jump.'''

    winWidth = 290
    winHeight = 290
```

```
    win = GraphWin('Ball Bounce', winWidth, winHeight)
    win.setCoords(0, 0, winWidth, winHeight)

    radius = 10
    xLow = radius # center is separated from the wall by the radius at a bounce
    xHigh = winWidth - radius
    yLow = radius
    yHigh = winHeight - radius

    center = getRandomPoint(xLow, xHigh, yLow, yHigh)
    ball = makeDisk(center, radius, win)

    bounceInBox(ball, dx, dy, xLow, xHigh, yLow, yHigh)
    win.close()

bounceBall(3, 5)
```

3.1.7. Compound Boolean Expressions. To be eligible to graduate from Loyola University Chicago, you must have 128 units of credit *and* a GPA of at least 2.0. This translates directly into Python as a compound condition:

```
units >= 128 and GPA >=2.0
```

This is true if both `units >= 128` is true and `GPA >=2.0` is true. A short example program using this would be:

```
units = input('How many units of credit do you have? ')
GPA = input('What is your GPA? ')
if units >= 128 and GPA >=2.0:
    print('You are eligible to graduate!')
else:
    print('You are not eligible to graduate.')
```

The new Python syntax is for the operator `and`:

condition1 `and` *condition2*

It is true if both of the conditions are true. It is false if at least one of the conditions is false.

Exercise 3.1.7.1. A person is eligible to be a US Senator who is at least 30 years old and has been a US citizen for at least 9 years. Write a version of a program `congress.py` to obtain age and length of citizenship from the user and print out if a person is eligible to be a Senator or not. A person is eligible to be a US Representative who is at least 25 years old and has been a US citizen for at least 7 years. Elaborate your program `congress.py` so it obtains age and length of citizenship and prints whether a person is eligible to be a US Representative only, or is eligible for both offices, or is eligible for neither.

In the last example in the previous section, there was an if-elif statement where both tests had the same block to be done if the condition was true:

```
if x < xLow:
    dx = -dx
elif x > xHigh:
    dx = -dx
```

There is a simpler way to state this in a sentence: If x < xLow or x > xHigh, switch the sign of dx. That translates directly into Python:

```
if x < xLow or x > xHigh:
    dx = -dx
```

The word `or` makes another compound condition:

condition1 `or` *condition2*

is true if *at least* one of the conditions is true. It is false if both conditions are false. This corresponds to *one* way the word "or" is used in English. Other times in English "or" is used to mean *exactly one* alternative is true.

It is often convenient to encapsulate complicated tests inside a function. Think how to complete the function starting:

```
def isInside(rect, point):
    '''Return True if the point is inside the Rectangle rect.'''

    pt1 = rect.getP1()
    pt2 = rect.getP2()
```

Recall that a `Rectangle` is specified in its constructor by two diagonally oppose `Points`. This example gives the first use in the tutorials of the `Rectangle` methods that recover those two corner points, `getP1` and `getP2`. The program calls the points obtained this way `p1` and `p2`. The x and y coordinates of `pt1`, `pt2`, and `point` can be recovered with the methods of the `Point` type, `getX()` and `getY()`.

Suppose that I introduce variables for the x coordinates of `p1`, `point`, and `p2`, calling these x-coordinates `end1`, `val`, and `end2`, respectively. On first try you might decide that the needed mathematical relationship to test is

```
end1 <= val <= end2
```

Unfortunately, this is not enough: The only requirement for the two corner points is that they be diagonally opposite, not that the coordinates of the second point are higher than the corresponding coordinates of the first point. It could be that `end1` is 200; `end2` is 100, and `val` is 120. In this latter case `val` *is* between `end1` and `end2`, but substituting into the expression above

```
200 <= 120 <= 100
```

is False. The 100 and 200 need to be reversed in this case. This makes a complicated situation, and an issue which must be revisited for both the x and y coordinates. I introduce an auxiliary function `isBetween` to deal with one coordinate at a time. It starts:

```
def isBetween(val, end1, end2):
    '''Return True if val is between the ends.
    The ends do not need to be in increasing order.'''
```

Clearly this is true if the original expression, `end1 <= val <= end2`, is true. You must also consider the possible case when the order of the ends is reversed: `end2 <= val <= end1`. How do we combine these two possibilities? The Boolean connectives to consider are ***and*** and *or*. Which applies? You only need one to be true, so ***or*** is the proper connective:

A correct but redundant function body would be:

```
if end1 <= val <= end2 or end2 <= val <= end1:
    return True
else:
    return False
```

Check the meaning: if the compound expression is True, return True. If the condition is False, return False – in either case return the *same* value as the test condition. See that a much simpler and neater version is to just return the value of the condition itself!

```
return end1 <= val <= end2 or end2 <= val <= end1
```

In general you should *not* need an `if-else` statement to choose between true and false values!

A side comment on expressions like

```
end1 <= val <= end2
```

Other than the two-character operators, this is like standard math syntax, chaining comparisons. In Python any number of comparisons can be *chained* in this way, closely approximating mathematical notation. Though this is good Python, be aware that if you try other high-level languages like Java and C++, such an expression is gibberish. Another way the expression can be expressed (and which translates directly to other languages) is:

```
end1 <= val and val <= end2
```

So much for the auxiliary function `isBetween`. Back to the `isInside` function. You can use the isBetween function to check the x coordinates, `isBetween(point.getX(), p1.getX(), p2.getX())`, and to check the y coordinates, `isBetween(point.getY(), p1.getY(), p2.getY())`. Again the question arises: how do you combine the two tests?

In this case we need the point to be both between the sides *and* between the top and bottom, so the proper connector is *and*.

Think how to finish the `isInside` method. Hint: [3]

Sometimes you want to test the opposite of a condition. As in English you can use the word `not`. For instance, to test if a Point was not inside Rectangle Rect, you could use the condition

```
not isInside(rect, point)
```

In general,

```
not condition
```

is `True` when *condition* is `False`, and `False` when *condition* is `True`.

The example program chooseButton1.py, shown below, is a complete program using the `isInside` function in a simple application, choosing colors. Pardon the length. Do check it out. It will be the starting point for a number of improvements that shorten it and make it more powerful in the next section. First a brief overview:

The program includes the functions isBetween and isInside that have already been discussed. The program creates a number of colored rectangles to use as buttons and also as picture components. Aside from specific data values, the code to create each rectangle is the same, so the action is encapsulated in a function, `makeColoredRect`. All of this is fine, and will be preserved in later versions.

The present main function is long, though. It has the usual graphics starting code, draws buttons and picture elements, and then has a number of code sections prompting the user to choose a color for a picture element. Each code section has a long if-elif-else test to see which button was clicked, and sets the color of the picture element appropriately.

```
'''Make a choice of colors via mouse clicks in Rectangles --
A demonstration of Boolean operators and Boolean functions.'''

from graphics import *

def isBetween(x, end1, end2):  #same as before
    '''Return True if x is between the ends or equal to either.
    The ends do not need to be in increasing order.'''

    return end1 <= x <= end2 or end2 <= x <= end1

def isInside(point, rect):
    '''Return True if the point is inside the Rectangle rect.'''

    pt1 = rect.getP1()
    pt2 = rect.getP2()
    return isBetween(point.getX(), pt1.getX(), pt2.getX()) and \
           isBetween(point.getY(), pt1.getY(), pt2.getY())

def makeColoredRect(corner, width, height, color, win):
    ''' Return a Rectangle drawn in win with the upper left corner
    and color specified.'''

    corner2 = corner.clone()
    corner2.move(width, -height)
    rect = Rectangle(corner, corner2)
    rect.setFill(color)
```

[3] Once again, you are calculating and returning a Boolean result. You do not need an `if-else` statement.

```
    rect.draw(win)
    return rect

def main():
    winWidth = 400
    winHeight = 400
    win = GraphWin('pick Colors', winWidth, winHeight)
    win.setCoords(0, 0, winWidth, winHeight)

    redButton = makeColoredRect(Point(310, 350), 80, 30, 'red', win)
    yellowButton = makeColoredRect(Point(310, 310), 80, 30, 'yellow', win)
    blueButton = makeColoredRect(Point(310, 270), 80, 30, 'blue', win)

    house = makeColoredRect(Point(60, 200), 180, 150, 'gray', win)
    door = makeColoredRect(Point(90, 150), 40, 100, 'white', win)
    roof = Polygon(Point(50, 200), Point(250, 200), Point(150, 300))
    roof.setFill('black')
    roof.draw(win)

    msg = Text(Point(winWidth/2, 375),'Click to choose a house color.')
    msg.draw(win)
    pt = win.getMouse()
    if isInside(pt, redButton):
        color = 'red'
    elif isInside(pt, yellowButton):
        color = 'yellow'
    elif isInside(pt, blueButton):
        color = 'blue'
    else :
        color = 'white'
    house.setFill(color)

    msg.setText('Click to choose a door color.')
    pt = win.getMouse()
    if isInside(pt, redButton):
        color = 'red'
    elif isInside(pt, yellowButton):
        color = 'yellow'
    elif isInside(pt, blueButton):
        color = 'blue'
    else :
        color = 'white'
    door.setFill(color)

    msg.setText('Click anywhere to quit.')
    win.getMouse()
    win.close()

main()
```

The only further new feature used is in the long return statement in `isInside`.

```
    return isBetween(point.getX(), pt1.getX(), pt2.getX()) and \
           isBetween(point.getY(), pt1.getY(), pt2.getY())
```

Recall that Python is smart enough to realize that a statement continues to the next line if there is an unmatched pair of parentheses or brackets. Above is another situation with a long statement, but there are no unmatched parentheses on a line. For readability it is best *not* to make an enormous long line that would run off your screen or paper. Continuing to the next line is recommended. You can make the *final* character on a line be a backslash (\) to indicate the statement continues on the next line. This is not particularly neat, but it is a rather rare situation. Most statements fit neatly on one line, and the creator of Python decided it was best to make the syntax simple in the most common situation. (Many other languages require a special statement terminator symbol like ';' and pay no attention to newlines).

The chooseButton1.py program is long partly because of repeated code. The next section gives another version involving lists.

3.2. Loops and Tuples

This section will discuss several improvements to the chooseButton1.py program from the last section that will turn it into example program chooseButton2.py.

First an introduction to *tuples*, which we will use for the first time in this section:

A tuple is similar to a list except that a literal tuple is enclosed in parentheses rather than square brackets and a tuple is immutable. In particular you cannot change the length or substitute elements, unlike a list. Examples are

```
(1, 2, 3)
('yes', 'no')
```

They are another way to make several items into a single object. You can refer to individual parts with indexing, like with lists, but a more common way is with multiple assignment. A silly simple example:

```
tup = (1, 2)
(x, y) = tup
print(x)  # printe 1
print(y)  # prints 2
```

Now back to improving the chooseButton1.py program, which has similar code repeating in several places. Imagine how much worse it would be if there were more colors to choose from and more parts to color!

First consider the most egregious example:

```
    if isInside(pt, redButton):
        color = 'red'
    elif isInside(pt, yellowButton):
        color = 'yellow'
    elif isInside(pt, blueButton):
        color = 'blue'
    else :
        color = 'white'
```

Not only is this exact `if` statement repeated several times, all the conditions within the `if` statement are very similar! Part of the reason I did not put this all in a function was the large number of separate variables. On further inspection, the particular variables `redButton`, `yellowButton`, `blueButton`, all play a similar role, and their names are not really important, it is their *associations* that are important: that redButton goes with 'red', ... When there is a sequence of things all treated similarly, it suggests a list and a loop. An issue here is that the changing data is *paired*, a rectangle with a color string. There are a number of ways to handle such associations. A very neat way in Python to package a pair (or more things together) is a *tuple*, turning several things into one object, as in (redButtton, 'red'). Objects such are this tuple can be put in a larger list:

```
choicePairs = [(redButtton, 'red'), (yellowButton, 'yellow'), (blueButton, 'blue')]
```

Such tuples may be neatly handled in a `for` statement. You can imagine a function to encapsulate the color choice starting:

```
def getChoice(choicePairs, default, win):
    '''Given a list choicePairs of tuples, with each tuple in the form
```

```
    (rectangle, choice), return the choice that goes with the rectangle
    in win where the mouse gets clicked, or return default if the click
    is in none of the rectangles.'''

    point = win.getMouse()
    for (rectangle, choice) in choicePairs:
        #....
```

This is the first time we have had a `for`-loop going through a list of *tuples*. Recall that we can do multiple assignments at once with tuples. This also works in a `for`-loop heading. The `for`-loop goes through one tuple in the list choicePairs at a time. The first time through the loop the tuple taken from the list is `(redButtton, 'red')`. This `for`-loop does a multiple assignment to `(rectangle, choice)`each time through the loop, so the first time `rectangle` refers to `redButton` and `choice` refers to `'red'`. The next time through the loop, the second tuple from the list is used, `(yellowButton, 'yellow')`so this time inside the loop `rectangle` will refer to `yellowButton` and `choice` refers to `'yellow'`....This is a neat Python feature.[4]

There is still a problem. We could test each rectangle in the for-each loop, but the original `if-elif`... statement in chooseButton1.py stops when the *first* condition is true. For-each statements are designed to go all the way *through* the sequence. There is a simple way out of this in a function: A `return` statement always stops the execution of a function. When we have found the rectangle containing the point, the function can `return` the desired choice immediately!

```
def getChoice(choicePairs, default, win):
    '''Given a list of tuples (rectangle, choice), return the choice
    that goes with the rectangle in win where the mouse gets clicked,
    or return default if the click is in none of the rectangles.'''

    point = win.getMouse()
    for (rectangle, choice) in choicePairs:
        if isInside(point, rectangle):
            return choice
    return default
```

Note that the else part in chooseButton1.py corresponds to the statement *after* the loop above. If execution gets past the loop, then none of the conditions tested in the loop was true.

With appropriate parameters, the looping function is a complete replacement for the original `if-elif` statement! The replacement has further advantages.

- There can be an arbitrarily long list of pairs, and the exact same code works.
- This code is clearer and easier to read, since there is no need to read through a long sequence of similar `if-elif` clauses.
- The names of the rectangles in the tuples in the list are never referred to. They are unnecessary here. Only a list needs to be specified. That could be useful earlier in the program ....

Are individual names for the rectangles needed earlier? No, the program only need to end up with the pairs of the form (rectangle, color) in a list. The statements in the original program, below, have a similar form which will allow them to be rewritten:

```
    redButton = makeColoredRect(Point(310, 350), 80, 30, 'red', win)
    yellowButton = makeColoredRect(Point(310, 310), 80, 30, 'yellow', win)
    blueButton = makeColoredRect(Point(310, 270), 80, 30, 'blue', win)
```

As stated earlier, we could use the statements above and then make a list of pairs with the statement

```
choicePairs = [(redButtton, 'red'), (yellowButton, 'yellow'), (blueButton, 'blue')]
```

[4]Particularly in other object-oriented languages where lists and tuples are way less easy to use, the preferred way to group associated objects, like rectangle and choice, is to make a custom object type containing them all. This is also possible and often useful in Python. In some relatively simple cases, like in the current example, use of tuples can be easier to follow, though the approach taken is a matter of taste. The topic of creating custom type of objects will not be taken up in these tutorials.

Now I will look at an alternative that would be particularly useful if there were considerably more buttons and colors.

All the assignment statements with `makeColorRect` have the same format, but differing data for several parameters. I use that fact in the alternate code:

```
choicePairs = list()
buttonSetup = [(310, 350, 'red'), (310, 310, 'yellow'), (310, 270, 'blue')]
for (x, y, color) in buttonSetup:
   button = makeColoredRect(Point(x, y), 80, 30, color, win)
   choicePairs.append((button, color))
```

I extract the changing data from the creation of the rectangles into a list, `buttonSetup`. Since more than one data items are different for each of the original lines, the list contains a tuple of data from each of the original lines. Then I loop through this list and not only create the rectangles for each color, but also accumulates the (rectangle, color) pairs for the list `choicePairs`.

Note the double parentheses in the last line of the code. The outer ones are for the method call. The inner ones create a single tuple as the parameter.

Assuming I do not need the original individual names of the Rectangles, this code with the loop will completely substitute for the previous code with its separate lines with the separate named variables and the recurring formats.

This code has advantages similar to those listed above for the `getChoice` code.

Now look at what this new code means for the interactive part of the program. The interactive code directly reduces to

```
msg = Text(Point(winWidth/2, 375),'Click to choose a house color.')
msg.draw(win)
color = getChoice(colorPairs, 'white', win)
house.setFill(color)

msg.setText('Click to choose a door color.')
color = getChoice(colorPairs, 'white', win)
door.setFill(color)
```

In the original version with the long `if-elif` statements, the interactive portion only included portions for the user to set the color of two shapes in the picture (or you would have been reading code forever). Looking now at the similarity of the code for the two parts, we can imagine another loop, that would easily allow for many more parts to be colored interactively.

There are still several differences to resolve. First the message `msg` is *created* the first time, and only the text is *set* the next time. That is easy to make consistent by splitting the first part into an initialization and a separate call to `setText` like in the second part:

```
msg = Text(Point(winWidth/2, 375),'')
msg.draw(win)

msg.setText('Click to choose a house color.')
```

Then look to see the differences between the code for the two choices. The `shape` object to be colored and the name used to describe the shape change: two changes in each part. Again tuples can store the changes of the form (shape, description). This is another place appropriate for a loop driven by tuples.. The (shape, description) tuples should be explicitly written into a list that can be called `shapePairs`. We could easily extend the list shapePairs to allow more graphics objects to be colored. In the code below, the roof is added.

The new interactive code can start with:

```
shapePairs = [(house, 'house'), (door, 'door'), (roof, 'roof')]
msg = Text(Point(winWidth/2, 375),'')
msg.draw(win)
for (shape, description) in shapePairs:
    prompt = 'Click to choose a + description + ' color.'
    # ....
```

Can you finish the body of the loop? Look at the original version of the interactive code. When you are done thinking about it, go on to my solution. The entire code is in example program `chooseButton2.py`, and also below. The changes from chooseButton1.py are in three blocks, each labeled #NEW in the code. The new parts are the `getChoice` function and the two new sections of `main` with the loops:

```
'''Make a choice of colors via mouse clicks in Rectangles --
Demonstrate loops using lists of tuples of data.'''

from graphics import *

def isBetween(x, end1, end2):
    '''Return True if x is between the ends or equal to either.
    The ends do not need to be in increasing order.'''

    return end1 <= x <= end2 or end2 <= x <= end1

def isInside(point, rect):
    '''Return True if the point is inside the Rectangle rect.'''

    pt1 = rect.getP1()
    pt2 = rect.getP2()
    return isBetween(point.getX(), pt1.getX(), pt2.getX()) and \
           isBetween(point.getY(), pt1.getY(), pt2.getY())

def makeColoredRect(corner, width, height, color, win):
    ''' Return a Rectangle drawn in win with the upper left corner
    and color specified.'''

    corner2 = corner.clone()
    corner2.move(width, -height)
    rect = Rectangle(corner, corner2)
    rect.setFill(color)
    rect.draw(win)
    return rect

def getChoice(choicePairs, default, win):   #NEW, discussed above
    '''Given a list of tuples (rectangle, choice), return the choice
    that goes with the rectangle in win where the mouse gets clicked,
    or return default if the click is in none of the rectangles.'''

    point = win.getMouse()
    for (rectangle, choice) in choicePairs:
        if isInside(point, rectangle):
            return choice
    return default

def main():
    winWidth = 400
    winHeight = 400
    win = GraphWin('Pick Colors', winWidth, winHeight)
    win.setCoords(0, 0, winWidth, winHeight)

    #NEW, shown in the discussion above
    choicePairs = list() # elements (button rectangle, color)
    buttonSetup = [(310, 350, 'red'), (310, 310, 'yellow'), (310, 270, 'blue')]
```

```
    for (x, y, color) in buttonSetup:
       button = makeColoredRect(Point(x, y), 80, 30, color, win)
       choicePairs.append((button, color))

    house = makeColoredRect(Point(60, 200), 180, 150, 'gray', win)
    door = makeColoredRect(Point(90, 150), 40, 100, 'white', win)
    roof = Polygon(Point(50, 200), Point(250, 200), Point(150, 300))
    roof.setFill('black')
    roof.draw(win)

    #NEW started in the discussion above.
    shapePairs = [(house, 'house'), (door, 'door'), (roof, 'roof')]
    msg = Text(Point(winWidth/2, 375),'')
    msg.draw(win)
    for (shape, description) in shapePairs:
        prompt = 'Click to choose a + description + ' color.'
        msg.setText(prompt)
        color = getChoice(choicePairs, 'white', win)
        shape.setFill(color)

    msg.setText('Click anywhere to quit.')
    win.getMouse()
    win.close()

main()
```

Run it.

With the limited number of choices in chooseButton1.py, the change in length to convert to chooseButton2.py is not significant, but the change in organization is significant if you try to extend the program, as in the exercise below. See if you agree!

Exercise 3.2.0.2. a. Write a program `chooseButton3.py`, modifying `chooseButton2.py`. Look at the format of the list `buttonSetup`, and extend it so there is a larger choice of buttons and colors. Add at least one button and color.

b. Further extend the program `chooseButton3.py` by adding some further graphical object shape to the picture, and extend the list `shapePairs`, so they can all be interactively colored.

c. (Optional) If you would like to carry this further, also add a prompt to change the outline color of each shape, and then carry out the changes the user desires.

d. (Optional Challenge) Look at the pattern within the list `buttonSetup`. It has a consistent x coordinate, and there is a regular pattern to the change in the y coordinate (a consistent decrease each time). The only data that is arbitrary each time is the sequence of colors. Write a further version `chooseButton4.py` with a function makeButtonSetup, that takes a list of color names as a parameter and uses a loop to create the list used as `buttonSetup`. End by returning this list. Use the function to initialize `buttonSetup`. If you like, make the function more general and include parameters for the x coordinate, the starting y coordinate and the regular y coordinate change.

3.3. While Statements

3.3.1. Simple `while` Loops. Other than the trick with using a `return` statement inside of a `for` loop, all of the loops so far have gone all the way through a *specified* list. In any case the `for` loop has required the use of a specific list. This is often too restrictive. A Python `while` loop behaves quite similarly to common English usage. If I say

> While your tea is too hot, add a chip of ice.

Presumably you would test your tea. If it were too hot, you would add a little ice. If you test again and it is still too hot, you would add ice again. *As long as* you tested and found it was true that your tea was too hot, you would go back and add more ice. Python has a similar syntax:

```
while condition :
    indentedBlock
```

Setting up the English example in a similar format would be:

```
while your tea is too hot :
    add a chip of ice
```

To make things concrete and numerical, suppose the following: The tea starts at 115 degrees Fahrenheit. You want it at 112 degrees. A chip of ice turns out to lower the temperature one degree each time. You test the temperature each time, and also print out the temperature before reducing the temperature. In Python you could write and run the code below, saved in example program cool.py:

```
temperature = 115                  #1
while temperature > 112:           #2
    print(temperature)             #3
    temperature = temperature - 1  #4
print('The tea is cool enough.')   #5
```

I added a final line after the while loop to remind you that execution follows sequentially after a loop completes.

If you play computer and follow the path of execution, you could generate the following table. Remember, that each time you reach the end of the indented block after the `while` heading, execution returns to the `while` heading:

line	temperature	comment
1	115	
2		115 > 112 is true, do loop
3		prints 115
4	114	115 - 1 is 114, loop back
2		114 > 112 is true, do loop
3		prints 114
4	113	114 - 1 is 113, loop back
2		113 > 112 is true, do loop
3		prints 113
4	112	113 - 1 is 112, loop back
2		112 > 112 is false, skip loop
5		prints that the tea is cool

Each time the end of the indented loop body is reached, execution returns to the `while` loop heading for another test. When the test is finally false, execution jumps past the indented body of the `while` loop to the next sequential statement.

A `while` loop generally follows the pattern of the successive modification loop introduced with for-each loops:

```
initialization
while continuationCondition:
    do main action to be repeated
    prepare variables for the next time through the loop
```

Test yourself: Figure out by following the code, what is printed?

```
i = 4
while (i < 9):
    print(i)
    i = i+2
```

Check yourself by running the example program `testWhile.py`.

In Python, `while` is not used *quite* like in English. In English you could mean to stop *as soon as* the condition you want to test becomes false. In Python the test is only made when execution for the loop starts,

not in the middle of the loop. *Predict* what will happen with this slight variation on the previous example, switching the order in the loop body. Follow it carefully, one step at a time.

```
i = 4           #1
while (i < 9): #2
    i = i+2    #3
    print(i)   #4
```

Check yourself by running the example program `testWhile2.py`.

The sequence order is important. The variable i is increased before it is printed, so the first number printed is 6. Another common error is to assume that 10 will *not* be printed, since 10 is *past* 9, but the test that may stop the loop is *not* made in the middle of the loop. Once the body of the loop is started, it continues to the end, even when `i` becomes 10.

line	i	comment
1	4	
2		4 < 9 is true, do loop
3	6	4+2=6
4		print 6
2		6 < 9 is true, do loop
3	8	6+2= 8
4		print 8
2		8 < 9 is true, do loop
3	10	8+2=10
4	112	print 10
2		10 < 9 is false, skip loop

Predict what happens in this related little program:

```
nums = list()
i = 4
while (i < 9):
    nums.append(i)
    i = i+2
print(nums)
```

Check yourself by running the example program `testWhile3.py`.

3.3.2. The range Function, In General. There is actually a much simpler way to generate the previous sequence, using a further variation of the `range` function. Enter these lines separately in the *Shell*. As in the simpler applications of `range`, the values are only generated one at a time, as needed. To see the entire sequence at once, convert the sequence to a list:

```
nums = list(range(4, 9, 2))
print(nums)
```

The third parameter is needed when the step size from one element to the next is not 1.

The most general syntax is

```
range(start, pastEnd, step)
```

The value of the second parameter is always *past* the final element of the list. Each element after the first in the list is step more than the previous one. Predict and try in the *Shell*:

```
list(range(4, 10, 2))
```

Actually the range function is even more sophisticated than indicated by the while loop above. The step size can be negative.

Try in the *Shell*:

```
list(range(10, 0, -1))
```

Do you see how 0 is *past* the end of the list?

Make up a `range` function call to generate the list of temperatures printed in the tea example, `115, 114, 113`. Test it in the *Shell*.

3.3.3. Interactive `while` Loops. The earlier examples of while loops were chosen for their simplicity. Obviously they could have been rewritten with range function calls. Now lets try a more interesting example. Suppose you want to let a user enter a sequence of lines of text, and want to remember each line in a list. This could easily be done with a simple repeat loop if you knew the number of lines to enter. For example if you want three lines:

```
lines = list()
print('Enter 3 lines of text')
for i in range(3):
    line = input('Next line: ')
    lines.append(line)

print('Your lines were:')  # check now
for line in lines:
    print(line)
```

The user may want to enter a bunch of lines and not count them all ahead of time. This means the number of repetitions would not be known ahead of time. A `while` loop is appropriate here. There is still the question of how to test whether the user wants to continue. An obvious but verbose way to do this is to ask before every line if the user wants to continue, as shown below and in the example file `readLines1.py`. Read it and then run it:

```
lines = list()
testAnswer = input('Press y if you want to enter more lines: ')
while testAnswer == 'y':
    line = input('Next line: ')
    lines.append(line)
    testAnswer = input('Press y if you want to enter more lines: ')

print('Your lines were:')
for line in lines:
    print(line)
```

The data must be initialized *before* the loop, in order for the first test of the while condition to work. Also the test must work when you loop back from the end of the loop body. This means the data for the test must also be set up a second time, *in* the loop body.

The `readLines1.py` code works, but it may be more annoying than counting ahead! Two lines must be entered for every one you actually want! A practical alternative is to use a *sentinel*: a piece of data that would *not* make sense in the regular sequence, and which is used to indicate the *end* of the input. You could agree to use the line DONE! Even simpler: if you assume all the real lines of data will actually have some text on them, use an *empty* line as a sentinel. (If you think about it, the Python Shell uses this approach when you enter a statement with an indented body.) This way you only need to enter one extra (very simple) line, no matter how many lines of real data you have.

What should the while condition be now? Since the sentinel is an empty line, you might think `line == ''`, but that is the *termination* condition, not the *continuation* condition: You need the *opposite* condition. To negate a condition in Python, you may use *`not`*, like in English,

```
not line == ''
```

. Of course in this situation there is a shorter way, `line != ''`. Run the example program `readLines2.py`, shown below:

```
lines = list()
print('Enter lines of text.')
print('Enter an empty line to quit.')
line = input('Next line: ')
while line != '':
```

```
        lines.append(line)
        line = input('Next line: ')

    print('Your lines were:')
    for line in lines:
        print(line)
```

Again the data for the test in the while loop heading must be initialized before the first time the `while` statement is executed and the test data must *also* be made ready inside the loop for the test after the body has executed. Hence you see the statements setting the variable `line` both before the loop and at the end of the loop body. It is easy to forget the second place inside the loop!

Comment the last line of the loop out, and run it again *after* reading the rest of this paragraph. It will never stop! The variable `line` will forever have the initial value you gave it! You actually can stop the program by entering CTRL-C. That means hold the CTRL key and press C.

As you finish coding a `while` loop, it is good practice to always double-check: Did I make a change to the variables, *inside* the loop, that will eventually make the loop condition false?

EXERCISE 3.3.3.1. a. Write a program `sumAll.py` that prompts the user to enter numbers, one per line, ending with a line containing 0, and keep a running sum of the numbers. At the end (only) print out the sum. You should not need to create a list! You can immediately increase the sum with each number entered.

3.3.4. Graphical Applications. Another place where a `while` loop could be useful is in interactive graphics. Suppose you want the user to be able to create a Polygon by clicking on vertices they choose interactively, but you do not want them to have to count the number of vertices ahead of time. A while loop is suggested for such a repetitive process. As with entering lines of text interactively, there is the question of how to indicate you are done (or how to indicate to continue). If you make only a certain region be allowed for the Polygon, then the sentinel can be a mouse click outside the region. The earlier interactive color choice example already has a method to check if a mouse click is inside a Rectangle, so that method can be copied and reused.

Creating a polygon is a unified activity with a defined product, so let's define a function. It involves a boundary rectangle and mouse clicks in a GraphWin, and may as well return the Polygon constructed. *Read* the following start:

```
def polyHere(rect, win):
    '''  Draw a polygon interactively in Rectangle rect, in GraphWin win.
    Collect mouse clicks inside rect into a Polygon.
    When a click goes outside rect, stop and return the final polygon.
    The polygon ends up drawn.  The method draws and undraws rect.'''

    # ....
```

It is useful to start by thinking of the objects needed, and give them names.

- A Polygon is needed. Call it `poly`.
- A list of vertices is needed. Call it `vertices`. I need to append to this list. It must be initialized first.
- The latest mouse click point is needed. Call it `pt`.

Certainly the overall process will be repetitious, choosing point after point. Still it may not be at all clear how to make an effective Python loop. In challenging situations like this it is often useful to imagine a concrete situation with a limited number of steps, so each step can be written in sequence without worrying about a loop.

For instance to get up to a triangle (3 vertices in our list and a fourth mouse click for the sentinel), you might imagine the following sequence, undrawing each old polygon before the next is displayed with the latest mouse click included:

```
rect.setOutline("red")
rect.draw(win)
vertices = list()
pt = win.getMouse()
```

```
    vertices.append(pt)
    poly = Polygon(vertices)
    poly.draw(win)          # with one point
    pt = win.getMouse()
    poly.undraw()
    vertices.append(pt)
    poly = Polygon(vertices)
    poly.draw(win)          # with two points
    pt = win.getMouse()
    poly.undraw()
    vertices.append(pt)
    poly = Polygon(vertices)
    poly.draw(win)          # with three points
    pt = win.getMouse()  # assume outside the region

    rect.undraw()
    return poly
```

There is a fine point here that *I* missed the first time. The vertices of a Polygon do not get mutated in this system. A new Polygon gets created each time with the new vertex list. The old Polygon does not go away automatically, and extraneous lines appear in the picture if the old polygon is not explicitly undrawn each time before a new version is redrawn with an extra vertex. The timing for the `undraw` needs to be after the next mouse click and presumably before the next Polygon is created, so it could be before or after the line `vertices.append(pt)`. I arbitrarily chose for it to go before the vertices list is changed . The rest of the order of the lines is pretty well fixed by the basic logic.

If you think of the repetitions through a large number of loops, the process is essentially circular (as suggested by the word 'loop'). The body of a loop in Python, however, is written as a *linear* sequence: one with a first line and a last line, a beginning and an end. We can cut a circle *anywhere* to get a piece with a beginning and an end. In practice, the place you cut the loop for Python has one main constraint. The continuation condition in the `while` heading must make sense there. The processing in Python from the end of one time through the loop to the beginning of the next loop is separated by the test of the condition in the heading.

It can help to look at a concrete example sequence like the steps listed above for creating a triangle. The continuation condition is for `pt` to be in the rectangle, so using the previously written function `isInside`, the loop heading will be

```
while isInside(pt, rect):
```

With this condition in mind, look for where to split to loop. It needs to be after a new `pt` is clicked (so it can be tested) and before the next Polygon is created (so it does not include the sentinel point by mistake). In particular, with the sequence above, look and see that the split could go before or after the `poly.undraw()` line. Exercise 3.3.4.1 below considers the case where the split goes before this line. I will proceed with the choice of splitting into a Python loop *after* the undraw line. This makes the loop be

```
    while isInside(pt, rect):
        vertices.append(pt)
        poly = Polygon(vertices)
        poly.draw(win)
        pt = win.getMouse()
        poly.undraw()
```

If you follow the total sequence of required steps above for making the concrete triangle, you see that this *full* sequence for the loop is only repeated twice. The last time there is no `poly.undraw()` step. I could redo the loop moving the undraw line to the top, which caused different issues (Exercise 3.3.4.1 below). Instead think how to make it work at the end of the final time through the loop.

There are several possible approaches. You want the `undraw` line every time except for the last time. Hence it is a statement you want sometimes and not others. That suggests an `if` statement. The times you want the `undraw` are when the loop will repeat again. This is the same as the continuation condition for the

loop, and you have just read the next value for `pt`! You could just add a condition in front of the last line of the loop:

```
if isInside(pt, rect):
    poly.undraw()
```

I find this option unaesthetic: it means duplicating the continuation test twice in every loop.

Instead of *avoiding* the `undraw` as you exit the loop, another option in this case is to *undo* it: just *re*draw the polygon one final time *beyond* the loop. This only needs to be done once, not repeatedly in the loop. Then the repetitious lines collapse neatly into the loop, leaving a few of lines of the overall sequence before and after the loop. In the end the entire function is:

```
def polyHere(rect, win):
    '''  Draw a polygon interactively in Rectangle rect, in GraphWin win.
    Collect mouse clicks inside rect into a Polygon.
    When a click goes outside rect, stop and return the final polygon.
    The polygon ends up drawn.  The method draws and undraws rect.'''

    rect.setOutline("red")
    rect.draw(win)
    vertices = list()
    pt = win.getMouse()
    while isInside(pt, rect):
        vertices.append(pt)
        poly = Polygon(vertices)
        poly.draw(win)
        pt = win.getMouse()
        poly.undraw()

    poly.draw(win) # undo the last poly.undraw()
    rect.undraw()
    return poly
```

Follow this code through, imagining three mouse clicks inside rect and then one click outside of rect. Compare the steps to the ones in the concrete sequence written out above and see that the match (aside from the last cancelling undraw and draw of poly).

This function is illustrated in a the example program `makePoly.py`. Other than standard graphics example code, the main program contains:

```
rect1 = Rectangle(Point(5, 55), Point(200, 120))
poly1 = polyHere(rect1, win)
poly1.setFill('green')

rect2 = Rectangle(Point(210, 50), Point(350, 350))
poly2 = polyHere(rect2, win)
poly2.setOutline('orange')
```

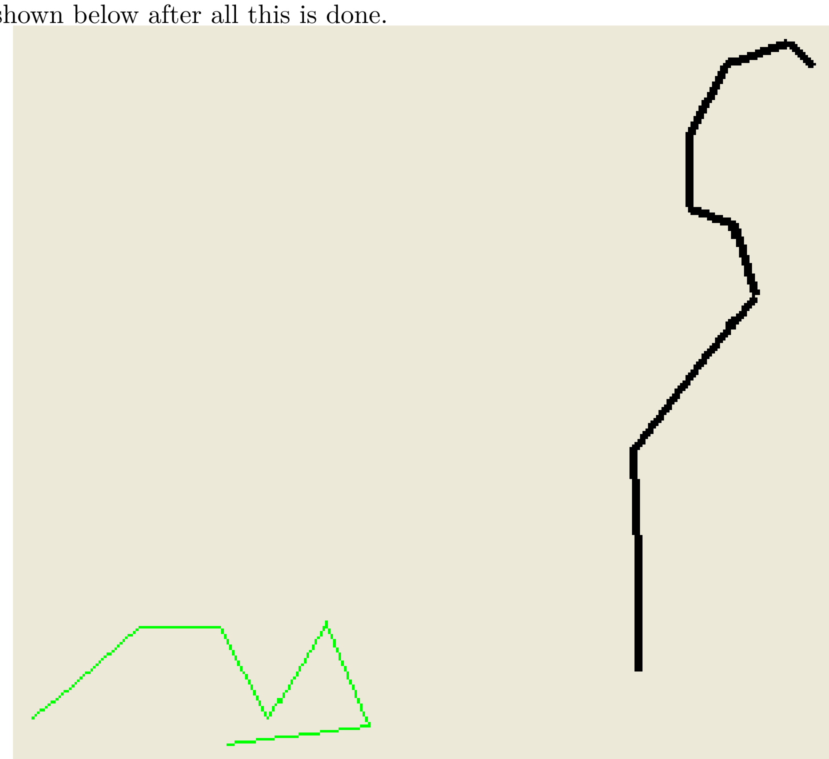
As you can see, the returned polygons are used to make color changes, just as an illustration.

EXERCISE 3.3.4.1. ** As discussed above, the basic loop logic works whether the `poly.undraw()` call is at the beginning or end of the loop. Write a variation `makePoly2.py` that makes the code work the other way, with the `poly.undraw()` at the beginning of the loop. The new place to cut the loop *does* affect the code before and after the loop. In particular, the extra statement drawing `poly` is not needed after the loop is completed. Make other changes to the surrounding code to make this work. Hints: [5]

[5] The basic issue is similar to the old version: the undraw is not always needed at the beginning, either. In this place it is not need the *first* time through the loop. The two basic approaches considered for the previous version still work here: break into cases inside the loop or make an extra compensating action outside the loop. Further hint: It is legal to draw a polygon with an empty vertex list – nothing appears on the screen.

Exercise 3.3.4.2. Write a program very similar to `makePoly.py`, and call it `makePath.py`, with a function `pathHere`. The only outward difference between `polyHere` and `pathHere` is that while the first creates a closed polygon, and returns it, and the new one creates a polygonal path, without the final point being automatically connected to the first point, and a list of the lines in the path is returned. Internally the functions are quite different. The change simplifies some things: no need to undraw anything in the main loop - just draw the latest segment each time going from the previous point to the just clicked point. There is a complication however, you do need deal specially with the first point. It has no previous point to connect to. I suggest you handle this before the main loop, and draw the point so it is a visible guide for the next point. After your main loop is finished undraw this initial point. (The place on the screen will still be visible if an initial segment is drawn. If no more points were added, the screen is left blank, which is the way it should be.) You also need to remember the previous point each time through the main loop.

In your main program, test the `makePath` function several times. Use the list of lines returned to loop and change the color in one path and the width of the lines in another path. A portion of a sample image is shown below after all this is done.



In earlier animation examples a `while` loop would also have been useful. Rather than continuing the animation a fixed number of times, it would be nice for the user to indicate by a mouse click when she has watched long enough. Thus far the only way to use the mouse has been with `getMouse()`. This is not going to work in an animation, because the computer stops and *waits* for a click with `getMouse()`, whereas the animation should *continue until* the click.

In full-fledged graphical systems that respond to events, this is no problem. Zelle's graphics is built on top of a capable event-driven system, and in fact, all mouse clicks are registered, even outside calls to `getMouse()`, though Zelle's documentation pages do not mention it.

As an example, run example program `randomCirclesWhile.py`. Be sure to follow the prompt saying to click to start *and* to end.

Aside from the prompts, the difference from the previous `randomCircles.py` program is the replacement of the original simple repeat loop heading

```
for i in range(75):
    code for an animation step
```

by the following initialization and while loop heading:

```
while win.checkMouse() == None:        #NEW
```

```
            code for an animation step
```

The graphics module remembers the last mouse click, whether or not it occurred during a call to `getMouse()`. An alternative way to check is `checkMouse()`. It does not *wait* for the mouse as in `getMouse()`. Instead it returns the *remembered* mouse click – the most recent mouse click in the *past*, unless there has been no mouse click since the last call to getMouse or checkMouse. In that case `checkMouse()` returns None (the special object used to indicate the lack of a regular object).

The `checkMouse` method allows for a loop that does not stop while waiting for a mouse click, but goes on until the heading test detects that the mouse *was* clicked.

A similar elaboration can be made for the other examples of animation, like `bounce1.py`. In `bounceWhile.py` I modified `bounce1.py` to have a while loop in place of the for-loop repeating 600 times. Run it. The only slight added modification here was that `win` was not originally a parameter to `bounceInBox`, so I included it. Look at the source code for `bounceWhile.py`, with the few changes marked NEW.

In `bounce2.py` I also made a more interesting change to the initialization, so the initial direction and speed of the mouse are determined graphically by the user, with a mouse click. Try example program `bounce2.py`.

The program includes a new utility function to help determine the initial (dx, dy) for the animation. This is done by calculating the move necessary to go from one point (where the ball is in this program) to another (specified by a user's mouse click in this program).

```
def getShift(point1, point2): # NEW utility function
    '''Returns a tuple (dx, dy) which is the shift from point1 to point2.'''
    dx = point2.getX() - point1.getX()
    dy = point2.getY() - point1.getY()
    return (dx, dy)
```

Since the function calculates both a change in x and y, it returns a tuple.

A straightforward interactive method, `getUserShift`, is wrapped around this function to get the user's choice, which ultimately returns the same tuple.

```
def getUserShift(point, prompt, win): #NEW direction selection
    '''Return the change in position from the point to a mouse click in win.
    First display the prompt string under point.'''

    text = Text(Point(point.getX(), 60), prompt)
    text.draw(win)
    userPt = win.getMouse()
    text.undraw()
    return getShift(point, userPt)
```

In the new version of the main driver, `bounceBall`, excerpted below, this interactive setting of (dx, dy) is used. Note the multiple assignment statement to both dx and dy, set from the tuple returned from `getUserShift`. This shift would generally be much too much for a single animation step, so the actual values passed to bounceBall are scaled way down by a factor `scale`.

```
    center = Point(winWidth/2, winHeight/2) #NEW central starting point
    ball = makeDisk(center, radius, win)

    #NEW interactive direction and speed setting
    prompt = '''
Click to indicate the direction and
speed of the ball:  The further you
click from the ball, the faster it starts.'''
    (dx, dy) = getUserShift(center, prompt, win)
    scale = 0.01 # to reduce the size of animation steps

    bounceInBox(ball, dx*scale, dy*scale, xLow, xHigh, yLow, yHigh, win)
```

The `bounceInBox` method has the same change to the loop as in the randomCircles.py example. The method then requires the `GraphWin`, `win`, as a further parameter, since `checkMouse` is a `GraphWin` method..

You can look in Idle at the full source code for `bounce2.py` if you like. The changes from `bounce1.py` are all marked with a comment starting with `#NEW`, and all the major changes have been described above.

In the examples so far of the use of `checkMouse()`, we have only used the fact that a point was clicked, not *which* point. The next example version, `bounce3.py`, does use the location of mouse clicks that are read with `checkMouse()` to change the direction and speed of the ball. Try it.

This version only slightly modifies the central animation function, `bounceInBox`, but wraps it in another looping function that makes the direction and speed of the ball change on *each* mouse click. Hence the mouse clicks detected in `bounceInBox` need to be remembered and then returned after the main animation loop finishes. That requires a name, `pt`, to be given to the last mouse click, so it can be remembered. This means modifying the main animation loop to initialize the variable `pt` before the loop and reset it at the end of the loop, much as in the use of getMouse() for the interactive polygon creation. That explains the first three NEW lines and the last two NEW lines in the revised `bounceInBox`:

```
def bounceInBox(shape, dx, dy, xLow, xHigh, yLow, yHigh, win):
    ''' Animate a shape moving in jumps (dx, dy), bouncing when
    its center reaches the low and high x and y coordinates.
    The animation stops when the mouse is clicked, and the
    last mouse click is returned.'''

    delay = .001
    pt = None                        #NEW
    while  pt == None:               #NEW
        shape.move(dx, dy)
        center = shape.getCenter()
        x = center.getX()
        y = center.getY()
        isInside = True              #NEW
        if x < xLow or x > xHigh:
            dx = -dx
            isInside = False         #NEW
        if y < yLow or y > yHigh:
            dy = -dy
            isInside = False         #NEW
        time.sleep(delay)
        if isInside: # NEW  don't mess with dx, dy when outside
            pt = win.checkMouse()  #NEW
    return pt                        #NEW
```

I initially made only the changes discussed so far (not the ones involving the new variable `isInside`). The variable `isInside` was in response to a bug that I will discuss after introducing the simple function that wraps around `bounceInBox`:

Each time the mouse is clicked, the ball is to switch direction and move toward the last click, until the stopping condition occurs, when there is a click above the stop line. This is clearly repetitive and needs a while loop. The condition is simply to test the y coordinate of the mouse click against the the height of the stop line. The body of the loop is very short, since we already have the utility function `getShift`, to figure out (dx, dy) values.

```
def moveInBox(shape, stopHeight, xLow, xHigh, yLow, yHigh, win): #NEW
    '''Shape bounces in win so its center stays within the low and high
    x and y coordinates, and changes direction based on mouse clicks,
    terminating when there is a click above stopHeight.'''
    scale = 0.01
    pt = shape.getCenter() # starts motionless
    while pt.getY() < stopHeight:
       (dx, dy) = getShift(shape.getCenter(), pt)
       pt = bounceInBox(shape, dx*scale, dy*scale,
```

```
xLow, xHigh, yLow, yHigh, win)
```

The variable `pt` for the last mouse click needed to be initialized some way. I chose to make the value be the same as the initial position of the ball, so both dx and dy are initially 0, and the ball does not start in motion. (Alternatives are in Exercise 3.3.4.3 below.)

I occasionally detected a bug when using the program. The ball would get stuck just outside the boundary and stay there. The fact that it was slightly beyond the boundary was a clue: For simplicity I had cheated, and allowed the ball to go just one animation step beyond the intended boundary. With the speed and small step size this works visually. The original code was *sure* to make an opposite jump back inside at the next step.

After some thought, I noticed that the initial version of the bounce3.py code for `bounceInBox` broke that assumption. When the ball was where a bounce-back is required, a mouse click could change (dx, dy) and mess up the bounce. The idea for a fix is not to let the user change the direction in the moment when the ball needs to bounce back.

Neither of the original boundary-checking `if` statements, by *itself*, always determines if the ball is in the region where it needs to reverse direction. I dealt with this situation by introducing a *Boolean variable* `isInside`. It is initially set as `True`, and then either of the `if` statements can correct it to False. Then, at the end of the loop, `isInside` is used to make sure the ball is safely inside the proper region when there is a check for a new mouse click and a possible user adjustment to (dx, dy).

EXERCISE 3.3.4.3. ** (Optional) I chose to have the ball start motionless, by making the initial value of `pt` (which determines the initial (dx, dy)) be the center of the ball. Write a variation `startRandom.py` so `pt` is randomly chosen. Also make the initial location of the ball be random. You can copy the function `getRandomPoint` from bounce1.py.

EXERCISE 3.3.4.4. ** Write a program `madlib4.py` that modifies the `getKeys` method of `madlib2.py` to use a `while` loop. (This is not an animation program, but this section is where you have had the most experience with while loops!) Hint: [6]

EXERCISE 3.3.4.5. ** Write a graphical game program, findHole.py, "Find the Hole". The program should use a random number generator to select a point and a perhaps radius around that point. These determine the target and are not revealed to the player initially. The user is then prompted to click around on the screen to "find the hidden hole". You should show the points the user has tried. Once the user selects a point that is within the chosen radius of the mystery point, the mystery circle should appear, and the point of the final successful mouse click should show. There should be a message announcing how many steps it took, and the game should end.

Hint: you have already seen the code to determine the displacement (dx, dy) between two points: use the `getShift` function in bounce2.py. Once you have the displacement (dx, dy) between the hidden center and the latest mouse click, the distance between the points is (dx*dx + dy*dy)**0.5, using the Pythagorean Theorem of geometry. If this distance is no more than the radius you have chosen for the mystery circle, then the user has found the circle! You can use `getShift` as written, or modify it into a function `getDistance` that directly returns the distance between two points.

Many elaborations on this game are possible! Have fun with it!

3.3.5. Fancier Animation Loop Logic (Optional). The final variation is the example program `bounce4.py`, which has the same outward behavior as bounce3.py, but it illustrates a different internal design decision. The bounce3.py version has two levels of while loop in two methods, `moveInBox`.for mouse clicks and `bounceInBox` for bouncing. The bounce4.py version puts all the code for changing direction inside the main animation loop in the old outer function, `moveInBox`. There are now three reasons to adjust (dx, dy): bouncing off the sides, bouncing off the top or bottom, or a mouse click. That is a simplification and unification of the logic in one sense. The complication now is that the logic for determining when to quit

[6] This is actually the most natural approach. I avoided `while` loops initially, when only `for` loops had been discussed. It is redundant in the original approach, however, to find every instance of `'{'` to *count* the number of repetitions and then *find* them all again when extracting the cue keys. A more natural way to control the loop is a `while` loop stopping when there are no further occurrences of `'{'`. This involves some further adjustments. You must cut the loop in a different place (to end after searching for `'{'`) . As discussed before, cutting a loop in a different place may require changes before and after the loop, too.

is buried deep inside the if-else logic, not at the heading of the loop. The test for mouse clicks is inside the while loop and further inside another if statment. The test of the mouse click may merely lead to a change in (dx, dy), or is a signal to quit. Here is the revised code, with a discussion afterward of the return statement:

```
def moveInBox(shape, stopHeight, xLow, xHigh, yLow, yHigh, win):
    ''' Animate a shape moving toward any mouse click below stopHeight and
    bouncing when its center reaches the low or high x or y coordinates.
    The animation stops when the mouse is clicked at stopHeight or above.'''

    scale = 0.01
    delay = .001
    dx = 0                              #NEW dx and dy no longer parameters
    dy = 0                              #NEW
    while True:                         #NEW exit loop at return statement
        center = shape.getCenter()
        x = center.getX()
        y = center.getY()
        isInside = True
        if x < xLow or x > xHigh:
            dx = -dx
            isInside = False
        if y < yLow or y > yHigh:
            dy = -dy
            isInside = False
        if isInside:
            pt = win.checkMouse()
            if pt != None:               #NEW dealing with mouse click now here
                if pt.getY() < stopHeight: # switch direction
                    (dx, dy) = getShift(center, pt)
                    (dx, dy) = (dx*scale, dy*scale)
                else:                    #NEW exit from depths of the loop
                    return               #NEW
        shape.move(dx, dy)
        time.sleep(delay)
```

Recall that a `return` statement immediately terminates function execution. In this case the function returns no value, but a bare `return` is legal to force the exit. Since the testing is not done in the normal `while` condition, the `while` condition is set as permanently `True`. This is not the most common `while` loop pattern! It obscures the loop exit. The choice between the approach of bounce3.py and bounce4.py is a matter of taste in the given situation.

3.4. Arbitrary Types Treated As Boolean

The following section would merely be an advanced topic, except for the fact that many common mistakes have their meaning changed and obscured by the Boolean syntax discussed.

You have seen how many kinds of objects can be converted to other types. *Any* object can be converted to Boolean (type bool). *Read* the examples shown in this Shell sequence:

```
>>> bool(2)
True
>>> bool(-3.1)
True
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool(None)
```

```
False
>>> bool('')
False
>>> bool('0')
True
>>> bool('False')
True
>>> bool([])
False
>>> bool([0])
True
```

The result looks pretty strange, but there is a fairly short general explanation: Almost everything is converted to `True`. The only values among built-in types that are interpreted as False are

- The Boolean value `False` itself
- Any numerical value equal to 0 (0, 0.0 but not 2 or -3.1)
- The special value `None`
- Any empty sequence or collection, including the empty string(`''`, but not `'0'` or `'hi'` or `'False'`) and the empty list (`[]`, but not `[1,2, 3]` or `[0]`)

A possibly useful consequence occurs in the fairly common situation where something needs to be done with a list only if it is nonempty. In this case the explicit syntax:

```
if len(aList) > 0:
    doSomethingWith(aList)
```

can be written with the more succinct python idiom

```
if aList:
    doSomethingWith(aList)
```

This automatic conversion can also lead to extra trouble! Suppose you prompt the user for the answer to a yes/no question, and want to accept 'y' or 'yes' as indicating True. You might write the following *incorrect* code. *Read* it:

```
ans = input('Is this OK? ')
if  ans == 'y' or 'yes':
    print('Yes, it is OK')
```

The problem is that there are two binary operations here: `==`, ***or***. Comparison operations all have higher precedence than the logical operations ***or***, ***and***. The `if` condition above can be rewritten equivalently with parentheses. *Read* and consider:

```
(ans == 'y') or 'yes'
```

Other programming languages have the advantage of stopping with an error at such an expression, since a string like `'yes'` is not Boolean. Python, however, accepts the expression, and treats `'yes'` as `True`! To test, *run* the example program `boolConfusion.py`, shown below:

```
ans = 'y'
if ans == 'y' or 'yes':
    print('y is OK')

ans = 'no'
if ans == 'y' or 'yes':
    print('no is OK!!???')
```

Python detects no error. The `or` expression is always treated as `True`, since `'yes'` is a non-empty sequence, interpreted as `True`.

The intention of the `if` condition presumably was something like

```
(ans == 'y') or (ans == 'yes')
```

This version also translates directly to other languages. Another correct Pythonic alternative that groups the alternate values together is

```
ans in ['y', 'yes']
```

which reads pretty much like English. It is true if `ans` is in the specified list. The `in` operator actually works with any sequence. The general syntax is

```
value in sequence
```

This is true when value is an element of the sequence.

Be careful to use a correct expression when you want to specifiy a condition like this.

Things get even stranger! Enter these conditions themselves, one at a time, directly into the *Shell*:

```
'y' == 'y' or 'yes'
'no' == 'y' or 'yes'
```

The meaning of `(a or b)` is exactly as discussed so far *if* each of the operands `a` and `b` are actually Boolean, but a more elaborate definition is needed if an operand is not Boolean.

```
val = a or b
```

means

```
if bool(a):
    val = a
else:
    val = b
```

and in a similar vein:

```
val = a and b
```

means

```
if bool(a):
    val = b
else:
    val = a
```

This strange syntax was included in Python to allow code like in the following example program `orNotBoolean.py`. *Read* and test if you like:

```
defaultColor = 'red'
userColor = input('Enter a color, or just press Enter for the default: ')
color = userColor or defaultColor
print('The color is', color)
```

which sets `color` to the value of `defaultColor` if the user enters an empty string.

Again, this may be useful to experienced programmers. The syntax can certainly cause difficult bugs, particularly for beginners!

The `not` operator always produces a result of type bool.

3.5. Further Topics to Consider

Chapter 4 gives an application of Python to the web. It does not introduce new language features. We have come to end of the new language features in this tutorial, but there are certainly more basic topics to learn about programming and Python in particular, if you continue in other places:

(1) Creating your own kinds of objects (writing classes)
(2) Inheritance: Building new classes derived from existing classes
(3) Python list indexing and slicing, both to read and change parts of lists
(4) Other syntax used with loops: `break`, `continue`, and `else`
(5) Exception handling
(6) Python's variable length parameter lists, and other options in parameter lists
(7) List comprehensions: a concise, readable, fast Pythonic way to make new lists from old ones
(8) Event handling in graphical programming
(9) Recursion (not special to Python): a powerful programming technique where functions call themselves

Beyond these language features, Python has a vast collection of useful modules. An example program, bbassign.py, is a real-world program that I have in regular use for processing inconveniently organized files created by Blackboard for homework submissions. It is a command-line script that uses string methods and slicing and both kinds of loops, as well is illustrating some useful componets in modules sys, os, and os.path, for accessing command line parameters, listing file directories, creating directories, and moving and renaming files.

3.6. Summary

(1) Comparison operators produce a Boolean result (type `bool`, either `True` or `False`): [3.1.4]

Meaning	Math Symbol	Python Symbols
Less than	$<$	`<`
Greater than	$>$	`>`
Less than or equal	$\leq$	`<=`
Greater than or equal	$\geq$	`>=`
Equals	$=$	`==`
Not equal	$\neq$	`!=`

Comparisons may be chained as in `a < b <= c < d != e`. [3.1.5]

(2) The `in` operator: [3.4]
`value` `in` *`sequence`*
is True if *value* is one of the elements in the *`sequence`*.

(3) Interpretation as Boolean (`True`, `False`):
All Python data may be converted to Boolean (type `bool`). The only data that have a Boolean meaning of `False`, in addition to `False` itself, are `None`, numeric values equal to 0, and empty collections or sequences, like the empty list `[]` and the empty string `''`. [3.4]

(4) Operators on Boolean expressions [3.1.7]
`condition1` `and` *`condition2`* True only if both conditions are True
`condition1` `or` *`condition2`* True only if at least one conditions is True
`not` *`condition`* True when *`condition`* is False; False when *`condition`* is True
This description is sufficient if the result is used as a Boolean value (in an `if` or `while` condition). See Section 3.4 for the advanced use when operands are not explicitly Boolean, and the result is not going to be interpreted as Boolean.

(5) `if` Statements

(a) Simple `if` statement [3.1.2]

```
if condition:
    indentedStatementBlockForTrueCondition
```

If the condition is true, then do the indented statement block. If the condition is not true, then s
If the condition is true, then do the first indented block only. If the condition is not true, then skip the first indented block and do the one after the `else:`.
If the condition is true, then do the first indented block only. If the condition is not true, then skip the first indented block and do the one after the `else:`.

(b) `if-else` statement [3.1.3]

```
if condition:
    indentedStatementBlockForTrueCondition
else:
    indentedStatementBlockForFalseCondition
```

If the condition is true, then do the first indented block only. If the condition is not true, then skip the first indented block and do the one after the `else:`.

(c) The most general syntax for an if statement, `if-elif-...-else` [3.1.5]:

```
if condition1 :
   indentedStatementBlockForTrueCondition1
```

```
elif condition2 :
    indentedStatementBlockForFirstTrueCondition2
elif condition3 :
  indentedStatementBlockForFirstTrueCondition3
elif condition4 :
    indentedStatementBlockForFirstTrueCondition4
else:
    indentedStatementBlockForEachConditionFalse
```

The `if`, each `elif`, and the final `else` line are all aligned. There can be any number of `elif` lines, each followed by an indented block. (Three happen to be illustrated above.) With this construction exactly *one* of the indented blocks is executed. It is the one corresponding to the *first* `True` condition, or, if all conditions are `False`, it is the block after the final `else` line

(d) `if-elif` [3.1.5]
The `else:` clause above may also be omitted. In that case, if none of the conditions is true, no indented block is executed.

(6) `while` statements [3.3.1]

```
while condition:
    indentedStatementBlock
```

Do the indented block if *condition* is `True`, and at the end of the indented block loop back and test the *condition* again, and continue repeating the indented block as long as the condition is `True` *after* completing the indented block. Execution does not stop in the middle of the block, even if the *condition* becomes `False` at that point.

A `while` loop can be used to set up an (intentionally) apparently infinite loop by making *condition* be just `True`. To end the loop in that case, there can be a test inside the loop that sometime becomes True, allowing the execution of a `return` statement to break out of the loop. [3.3.5]

(7) `range` function with three parameters [3.3.1]
`range(start, pastEnd, step)`
Return a list of elements `[start, start+step, ...]`, with each element *step* from the previousCGI one, ending just before reaching *pastEnd*. If *step* is positive, *pastEnd* is larger than the last element. If *step* is negative, *pastEnd* is smaller than the last element.

(8) Type `tuple`
`(` expression `,` expression `,` and so on `)`
`(` expression `,` `)`
`( )`
(a) A literal tuple, with two or more elements, consists of a comma separated collection of values all enclosed in parentheses. A literal tuple with only a single element must have a comma after the element to distinguish from a regular parenthesized expression. [3.2]
(b) A tuple is a kind of sequence.
(c) Tuples, unlike lists, are immutable (may not be altered)..

(9) Additional programming techniques
(a) These techniques extend the techniques listed in the summary of the previous chapter. [2.6]
(b) The basic pattern for programming with a while loop is [3.3.1]

```
initialization
while continuation condition :
    main action to repeat
    prepare variables for next loop
```

(c) Interactive while loops generally follow the pattern [3.3.3]

```
input first data from user
while continue based on test of user data :
    process user data
    input next user data
```

Often the code to input the first data and the later data is the same, but it must appear in both places!

(d) Sentinel Loops [3.3.3]
Often the end of the repetition of a data-reading loop is indicated by a *sentinel* in the data: a data value known to both the user and the program to not be regular data, that is specifically used to signal the end of the data.

(e) Nesting Control Flow Statements [3.1.6]
  (i) `If` statements may be nested inside loops, so the loop does not have to execute all the same code each time.
  (ii) Loops may be nested. The inner loop completes its repetitions for each time through the outer loop.

(f) Breaking a repeating pattern into a loop [3.3.4]
Since a loop is basically circular, there may be several choices of where to split it to list it in the loop body. The split point needs to be where the continuation test is ready to be run, but that may still allow flexibility. When you choose to change the starting point of the loop, and rotate statements between the beginning and the end of the loop, you change what statements need to be included before and after the loop, sometimes repeating or undoing actions taken in the loop.

(g) Tuples in lists [3.2]
A list may contain tuples. A for-each loop may process tuples in a list, and the for loop heading can do multiple assignments to variables for each element of the next tuple.

(h) Tuples as return values [3.2]
A function may return more than one value by wrapping them in a tuple. The function may then be used in a multiple assignment statement to extract each of the returned variables.

(10) Graphics
  (a) Zelle's Graphics GraphWin method `checkMouse()` allows mouse tests without stopping animation, by testing the last mouse click, not waiting for a new one. [3.3.4]
  (b) The most finished examples of using graphics.py are in [3.2] and [3.3.4]

CHAPTER 4

Dynamic Web Pages

This chapter leads up to the creation of dynamic web pages. These pages and supporting programs and data may be tested locally via a simple Python web server available on your local machine. If you have access, the pages and programs may be uploaded to a public server accessible to anyone on the Internet.

A few disclaimers:

- This tutorial does not cover uploading to an account on a public server.
- No core Python syntax is introduced in this Chapter. Only a few methods in a couple of Python library modules are introduced.
- The chapter is by no means a major source of information about HTML code. That is mostly avoided with the use of a modern word-processor-like HTML editor. As a specific example, the open source HTML editor Kompozer is discussed.

The chapter does allow you to understand the overall interaction between a browser (like Firefox on your local machine) and a web server and to create dynamic web content with Python. We treat interaction with the web basically as a mechanism to get input into a Python program and data back out and displayed. Web pages displayed in your browser are used for both the input and the output. The advantage of a public server is that it can also be used to store data accessible to people all over the world.

There are a number of steps in the development in this chapter, so I start with an overview:

(1) A few bits about the basic format of *hypertext markup language* are useful to start.
(2) The simplest pages to start writing in Kompozer are just *static web pages*, formatted like a word-processing document.
(3) Next we look at pages generated *dynamically*. An easy way to accomplish this is to create specialized static pages to act as templates into which the dynamic data is easily embedded. Web page creation can be tested totally locally, by creating HTML files and pointing your web browser to them. Initially we supply input data by our traditional means (keyboard input or function parameters), and concentrate on having our Python program *convert* the input to the desired output, and *display* this output in a web page.
(4) We generate data from within a web page, using web forms (generated via Kompozer). Initially we will test web forms by automatically dumping their raw data.
(5) To fully integrate a browser and server, we use 1) web forms to provide data, 2) a Python program specified on the server to transform the input data into the desired output, 3) embed the output in a new dynamic web page that gets sent back to your browser. This Python server program transforms the input data, and generates output web pages much like we did in step 3.
(6) Finally, if you have an account like Loyola Computer Science students, you can upload and show off your work on your own personal web site, accessible to everyone on the Internet.

4.1. Web page Basics

4.1.1. Format of Web Page Markup. Documents can be presented in many forms. A simple editor like Idle or Windows' Notepad produce plain text: essentially a long string of meaningful characters.

Documents can be displayed with formatting of parts of the document. Web pages allow different fonts, italic, and boldfaced emphases, and different sized text, all to be included. Microsoft Word, Open Office, and Latex, all display documents with various amounts of formatting. The syntax for the ways different systems *encode* the formatting information varies enormously.

If you look at a Microsoft Word document in a plain text editor like notepad, you should be able to find the original text buried inside, but most of the symbols associated with the formatting are unprintable gibberish as far as a human is concerned.

Hypertext markup language (HTML) is very different in that regard. It produces a file of entirely human-readable characters, that could be produced with a plain text editor.

For instance in HTML, the largest form of a heading with the text “Web Introduction”, would look like

<h1>Web Introduction</h1>

The heading format is indicated by bracketing the heading text ’Web Introduction’ with markup sequences, <h1> beforehand, and </h1> afterward. All HTML markup is delimited by *tags* enclosed in angle brackets, and most tags come in pairs, surrounding the information to be formatted. The end tag has an extra ’/’. Here ’h’ stands for heading, and the number indicates the relative importance of the heading. (There is also h2, h3, ... for smaller headings.) In the early days of HTML editing was done in a plain text editor, with the tags being directly typed in by people who memorized all the codes!

With the enormous explosion of the World Wide Web, specialized software has been developed to make web editing be much like word processing, with a graphical interface, allowing formatting to be done by selecting text with a mouse and clicking menus and icons labeled in more natural language. The software then automatically generates the necessary markup. An example used in these tutorials is the open source Kompozer, available at `http://kompozer.net`. (Careful – although this is free, open source software, the URL is Kompozer.*net*, not Kompozer.org. There is a site Kompozer.org that is *designed* to confuse you!) You can open Kompozer and easily generate a document with a heading, and italic and boldfaced portions....

4.1.2. Introduction to Static Pages in Kompozer. This section introduces the Kompozer web page editor to create static pages. Kompozer is used because it is free software, and is pretty easy to use, like a common word processor. Unlike a common word processor you will be able to easily look at the HTML markup code underneath. It is not necessary to know a lot about the details of the markup codes for HTML files to use Kompozer.

We will use static pages later as a part of making dynamic pages, using the static pages as templates in which we insert data dynamically.

To creating static web pages

(1) If you are in a Loyola University Windows lab, go to the start menu -> Loyola software -> Internet -> Kompozer. (It may be under Math and Comp Sci instead.) You may get pop-up window wanting to count users of Kompozer. Click OK as another user of Kompozer.
(2) However you start Kompozer, go to the File menu and click on New. You will get what looks like an empty document.
(3) Look at the bottom of your window. You should see a ’*Normal*’ tabs selected, with other choices beside it, including a *Source* tab. Click on the *Source* tab. You should see that, though you have added no content, you already have the basic markup to make an html page!
(4) Click again on the Normal tab to go back to the Normal view (of no content at the moment).
(5) Assume you are making a home page for yourself. Make a title and some introductory text. Use regular word processor features like marking your title as Heading 1 in the drop down box on a menu bar. (The drop down menu may start off displaying ’Paragraph’ or ’Body Text’.) You can select text and make it bold or italics; enlarge it ... using the editing menu or icons.
(6) Before getting too carried away, save your document as index.html in the existing *www directory under your earlier Python examples.* It will save a lot of trouble if you keep your web work together in this www directory, where I have already placed a number of files yo will want to keep together in one directory.
(7) Just for comparison, switch back and forth between the Normal and Source views to see all that has gone on underneath your view, particularly if you edited the format of your text. Somewhere embedded in the Source view you should see all the text you entered. Some individual characters have special symbols in HTML that start with an ampersand and end with a semicolon. Again, at this point it is more important the understand that there are two different views than to be able to reproduce the Source view from memory.
(8) You can use your web browser to see how your file looks outside the editor. The easiest way to do this is to go to the web browser’s File menu and Open File entry, and find the index.html file. It should look pretty similar to the way it looked in Kompozer, but if you put in hyperlinks, they should now be active.

The discussion of web page editing continues in Section 4.3.4, on html forms, but first we get Python into the act.

4.1.3. Editing and Testing Different Document Formats. In this chapter you will be working with several different types of documents that you will edit and test in very different ways. The ending of their names indicate their use. Each time a new type of file is discussed in later sections, the proper ways to work with it will be repeated, but with all the variations, it is useful to group them all in one place now:

...Web.py: *My* convention for regular Python programs taking all their input from the keyboard, and producing output displayed on a web page. These programs can be run like other Python programs, directly from an operating system folder or from inside Idle.

...html: Web documents most often composed in an editor like Kompozer. By *my* convention, these are split into two categories

- **...Template.html or ...Output.html:** are not intended to be displayed directly in a browser, but instead are read by a Python program (...cgi or ...Web.py) to create a template or format string for a final web page that is dynamically generated inside the Python program.
- **Other:** files ending in .html are intended to be directly viewed in a web browser. Except for the simple static earlier examples in Section 4.1.2, they are designed to reside on a web server, where they can pass information to a Python CGI program. To make this work on your computer:
  (1) Have all the web pages in the same directory as the example program localCGIServer.py
  (2) Have localCGIServer.py running, started from a directory window, *not from inside Idle*
  (3) In the browser URL field, the web page file name must be preceded by http://localhost:8080/. For example, http://localhost:8080/adder.html would refer to the file adder.html, in the same directory as the running localCGIServer.py.

...cgi: Python CGI programs, intended to be run from a web server. It is sometimes useful to access a CGI program directly in a browser. To run on your computer, like the Other html files above, you need to refer to a URL starting with http://localhost:8080/. For example, http://localhost:8080/now.cgi would call the file now.cgi, which must be in the same directory as the running localCGIServer.py. More often CGI programs are referenced in a web form, and the program is called indirectly by the web server. CGI programs can be edited and saved inside Idle, but they do *not* run properly from inside Idle.

4.2. Composing Web Pages in Python

4.2.1. Dynamically Created Static Local Pages from Python. For the rest of this chapter, the example files will come from the www directory under the main examples directory you unzipped. I will refer to example file there as "example www files".

As the overview indicated, dynamic web applications typically involve getting input in from a web page form, processing the input in a program on the server, and displaying output to a web page. Introducing all these new ideas at once could be a lot to absorb, so this section treats only the last part, output to the web, and uses familiar keyboard input into a regular Python program.

Follow this sequence of steps:

Open the example www file hello.html in your browser, to see what it looks like.

Change your browser view - for instance go back to the previous page you displayed.

Open the same hello.html file in Kompozer.

In Kompozer, switch to the Source view (clicking the Source tab). Sometimes you will want to copy HTML text into a Python program. For instance, I selected and copied the entire contents of the hello.html source view and pasted it into a multi line string in the Python program shown and discussed below.

Careful, note the change from past practice here: Start Python *from inside the www directory*. In Windows start the Idle shortcut link that I placed *in the www directory*, not the original example directory. Open the www example program `helloWeb1.py` in an Idle edit window.

Run it. You should see a familiar web page appear in your *default* browser (possibly not the one you have been using). This is obviously not a very necessary program, since you can select this page directly in your browser! Still, one step at a time, it illustrates several useful points. The program is copied below. *Read* it:

```
'''A simple program to create an html file from a given string,
and call the default web browser to display the file.'''

contents = '''<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta content="text/html; charset=ISO-8859-1"
 http-equiv="content-type">
  <title>Hello</title>
</head>
<body>
Hello, World!
</body>
</html>
'''

def main():
    browseLocal(contents, 'helloPython.html')

def strToFile(text, filename):
    """Write a file with the given name and the given text."""
    output = open(filename,"w")
    output.write(text)
    output.close()

def browseLocal(webpageText, filename):
    """Start your webbrowser on a local file containing the text."""
    strToFile(webpageText, filename)
    import webbrowser
    webbrowser.open(filename)

main()
```

This program encapsulates two basic operations into the last two functions that will be used *over and over*. The first, `strToFile`, has nothing new, it just puts specified text in a file with a specified name. The second, `browseLocal`, does more. It takes specified text (presumably a web page), puts it in a file, and directly displays the file in your web browser. It uses the `open` function from the `webbrowser` module to start the new page in your web browser.

In this particular program the text that goes in the file is just copied from the literal string named `contents` in the program.

This is no advance over just opening the file in the browser directly! Still, it is a start towards the aim of creating web content dynamically.

An early example in this tutorial displayed the fixed `'Hello World!'` to the screen. This was later modified in `hello_you4.py` to incorporate user input using the string format method of Section 1.12.2,

```
person = input('Enter your name: ')
greeting = 'Hello {person}!'.format(**locals())
print(greeting)
```

Similarly, I can turn the web page contents into a format string, and insert user data. Load and *run* the www example program `helloWeb2.py`.

The simple changes from `helloWeb1.py` are marked at the beginning of the file and shown below. I modified the web page text to contain `'Hello, {person}!'` in place of `'Hello, World!'`, making the string into a *format* string, which I renamed to the more appropriate `pageTemplate`. The changed initial portion with the literal string and and the main program then becomes

```
pageTemplate = '''
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta content="text/html; charset=ISO-8859-1"
 http-equiv="content-type">
  <title>Hello</title>
</head>
<body>
Hello, {person}!
</body>
</html> ''' # NEW note '{person}' two lines up

def main():
    person = input('Enter a name: ')  # NEW
    contents = pageTemplate.format(**locals())   # NEW
    browseLocal(contents, 'helloPython2.html') # NEW filename
```

incorporating the person's name into the contents for the web page before saving and displaying it.

In this case, I stored the literal format string inside the Python program, but consider a different approach:

Load and *run* the www example program `helloWeb3.py`. It behaves exactly like helloWeb2.py, but is slightly different internally – it does not directly contain the web page template string. Instead the web page template string is read from the file `helloTemplate.html`.

Below is the beginning of helloWeb3.py, showing the only new functions. The first, `fileToStr`, will be a standard function used in the future. It is the inverse of `strToFile`.

The main program obtains the input. In this simple example, the input is used directly, with little further processing. It is inserted into the web page, using 'helloTemplate.html' as a format string.

```
def fileToStr(fileName): # NEW
    """Return a string containing the contents of the named file."""
    fin = open(fileName);
    contents = fin.read();
    fin.close()
    return contents

def main():
    person = input('Enter a name: ')
    contents = fileToStr('helloTemplate.html').format(**locals())   # NEW
    browseLocal(contents, 'helloPython3.html') # NEW filename
```

Although helloTemplate.html is not intended to be viewed by the user (being a template), you should open it in a web editor (like Kompozer) to look at it. It is legal to create a web page in a web page editor with expressions in braces embedded in it! If you look in the source view in Kompozer you will see something similar to the literal string in helloWeb2.py, except the lines are broken up differently. (This makes no difference in the formatted result, since in html, a newline is treated the same way as a space.)

Back in the Normal mode, add some formatting like italics, and an extra line of text, and save the file again (under the *same* name). Run the program `helloWeb3.py` again, and see that you have been able to change the *appearance* of the output without changing the Python program itself. That is the aim of using the template html page, allowing the web output formatting to be managed mostly *independently* from the Python program.

A more complicated but much more common situation is where the input data is processed and transformed into results somehow, and these *results*, often along with some of the original input, are embedded in the web page that is produced.

As a simple example, load and ***run*** the www example program `additionWeb.py`, which uses the template file additionTemplate.html.

The aim in the end of this chapter is to have user input come from a form on the web rather than the keyboard on a local machine, but in either case the input is still transformed into results and all embedded in a web page. To make parts easily reusable, I *obtain* the input in a *distinct* place from where the input is *processed*. In keeping with the later situation with web forms, all input is of string type (using keyboard `input` for now).

Look at the program. You will see only a few new lines! Because of the modular design, most of the program is composed of recent standard functions reused.

The only new code is at the beginning and is shown here:

```
def processInput(numStr1, numStr2):  # NEW
    '''Process input parameters and return the final page as a string.'''
    num1 = int(numStr1) # transform input to output data
    num2 = int(numStr2)
    total = num1+num2
    return fileToStr('additionTemplate.html').format(**locals())

def main(): # NEW
    numStr1 = input('Enter an integer: ')  # obtain input
    numStr2 = input('Enter another integer: ')
    contents = processInput(numStr1, numStr2)   # process input into a page
    browseLocal(contents, 'helloPython3.html') # display page
```

The input is obtained (via `input` for now), and it is processed into a web page string, and as a separate step it is displayed in a local web page.

There are a few things to note:

- All input is strings. Before the numerical calculations, the digit strings must be converted to integers.
- I do calculate (a very simple!) result and use it in the output web page.
- Although it is not in the Python code, an important part of the result comes from the web page format string in additionTemplate.html, which includes the needed variable names in braces, {num1}, {num2}, and {total}.

EXERCISE 4.2.1.1. ** Save `additionWeb.py` as `quotientWeb.py`. Modify it to display the results of a division problem in a web page. You can take your calculations from Exercise 1.10.3.2. You should only need to make Python changes to the `processInput` and `main` functions. You will also need the HTML for the page displayed. Make a web page template file called quotientTemplate.html and read it into your program.

In this version generated by a regular Python program, the web page is just a file generated by the program, and then the file gets displayed in the browser. When you call browseLocal, you must supply a filename for the file to be created, but the name is totally arbitrary.

4.3. CGI - Dynamic Web Pages

CGI stands for Common Gateway Interface. This interface is used by web servers to process information requests supplied by a browser. Python has modules to allow programs to do this work. The convention used by many servers is to have the server programs that satisfy this interface end in '.cgi'. That is the convention used below. All files below ending in '.cgi' are CGI programs on a web server, and in this chapter, they will all be Python programs (though there are many other languages in use for this purpose). These programs are often called scripts, so we will be dealing with *Python CGI scripts*.

4.3.1. An Example in Operation. The first part of this section requires you to have access to the Internet. Later you will also see that you can illustrate the exact same actions on your own local machine.

For a very simple but complete example, use your browser to go to the page on the public Loyola server, `http://cs.luc.edu/anh/python/hands-on/examples/www/adder.html`. You see a web *form*. Follow the instructions, enter numbers, and click on the Find Sum button. You get back a page that obviously used *your* data. This is the idea that you can generalize. First consider the basic execution steps behind the scene:

(1) The data you type is handled directly by the browser. It recognizes forms.

(2) An action instruction is stored in the form saying what to do when you press a button indicating you are ready to process the data (the Find Sum button in this case).
(3) In the cases we consider in this tutorial, the action is given as a web resource, giving the location of a CGI script on some server (in our cases, the same directory on the server as the current web page).
(4) When you press the button, the browser sends the data that you entered to that web location (in this case adder.cgi in the same folder as the original web page).
(5) The server recognizes the web resource as an executable script, sees that it is a Python program, and executes it, using the data sent along from the browser form as input.
(6) The script runs, manipulates its input data into some results, and puts those results into the text of a web page that is the output of the program.
(7) The server captures this output from the program and sends it back to your browser as a new page to display.
(8) You see the results in your browser.

This also works *locally*, entirely on your own computer, using a simple server built into Python. (*Internet no longer needed!*)

In an operating system file window, go to the folder with the www examples. Double click on `localCGIServer.py` to start the local, internal, web server. You should see a console window pop up, saying "Localhost CGI server started" . Once the server is started, leave the console window there as long as you want the local server running. Do *not* start the local server running from inside Idle.

Caution: If the server aborts and gives an error message about spaces in the path, look at the path through the parent directories over this www directory. If any of the directory names have spaces in them, the local file server will not work. Either go up the directory chain and alter the directory names to eliminate spaces or move the examples directory to a directory that does not have this issue. In particular, you need to move your examples directory if it is under the 'My Programs' directory.

Back in the www directory,

(1) Open the web link `http://localhost:8080/adder.html` (preferably in a new window, separate from this this tutorial).
(2) You should see an adder form in your browser again. Note that the web address no longer includes 'cs.luc.edu'. Instead it starts with 'localhost:8080', to reference the local Python server you started. Fill out the form and test it as before.
(3) Look at the console window. You should see a log of the activity with the server. *Close* the server window.
(4) Reload the web link `http://localhost:8080/adder.html`. You should get an error, since you refer to localhost, but you just stopped the local server.

For the rest of this chapter, we will be wanting to use the local server, so *restart* localCGIServer.py, and *keep it going*.

4.3.2. A Simple Buildup. Before we get too complicated, consider the source code of a couple of even simpler examples.

4.3.2.1. *hellotxt.cgi.* The simplest case is a CGI script with no input. The script just generates plain text, rather than HTML. *Assuming you have your local server going*, you can go to the link for hellotxt.cgi, `http://localhost:8080/hellotxt.cgi`. The code is in the www example directory, `hellotxt.cgi`, and below for you to *read*:

```
#!/usr/bin/python

# Required header that tells the browser how to render the text.
print("Content-Type: text/plain\n\n") # here text -- not html

# Print a simple message to the display window.
print("Hello, World!\n")
```

The top line is what tells a Unix server that this is a Python program. It says where to find the Python interpreter to process the rest of the script. This exact line is always required to run on a Unix server (like

the one Loyola's Computer Science Department uses). The line is ignored in Windows. If you leave the line there as a part of your standard text, you have one less thing to think about when uploading to a Unix server.

The first print function is telling the server which receives this output, that the format of the rest of the output will be plain text. This information gets passed back to the browser later. This line should be included exactly as stated IF you only want the output to be plain text (the simplest case, but not our usual case).

The rest of the output (in this case just from one print function) becomes the body of the plain text document you see on your browser screen, verbatim since it is plain text. The server captures this output and redirects it to your browser.

4.3.2.2. *hellohtml.cgi.* We can make some variation and display an already determined *html* page rather than plain text. Try the link `http://localhost:8080/hellohtml.cgi`. The code is in the www example directory, `hellohtml.cgi`, and below for you to *read*:

```
#!/usr/bin/python

print("Content-Type: text/html\n\n") # html markup follows

print("""
<html>
  <Title>Hello in HTML</Title>
  <body>
    <p>Hello There!</p>
    <p><b>Hi There!</b></p>
  </body>
</html> """)
```

There are two noteworthy changes. The *first* print function now declares the rest of the output will be *html*. This is the standard line you will be using for your CGI programs. The remaining print function has the markup for an html page. Note that the enclosing triple quotes work for a multi line string. Other than as a simple illustration, this CGI script has no utility: Just putting the contents of the last print function in a file for a static web page hello.html is much simpler.

4.3.2.3. *now.cgi.* One more simple step: we can have a CGI script that generates dynamic output by reading the clock from inside of Python: Try the link `http://localhost:8080/now.cgi`. Then click the refresh button and look again. This cannot come from a static page. The code is in the www example directory, `now.cgi`, and below for you to *read*:

```
#!/usr/bin/python

import time
print("Content-Type: text/html\n\n") # html markup follows

timeStr = time.strftime("%c") # obtains complete current time

htmlFormat = """
<html>
  <Title>The Time Now</Title>
<body>
  <p>The current Central date and time is: {timeStr}</p>
</body>
</html> """

print(htmlFormat.format(**locals())) # see embedded {timeStr} ^ above
```

This illustrates a couple more ideas: First a library module, `time`, is imported and used to generate the string for the current date and time.

The web page is generated like in helloWeb2.py, embedding the dynamic data (in this case the time) into a literal web page format string. (Note the embedded `{timeStr}`.) Unlike helloWeb2.py, this is a CGI script so the web page contents are delivered to the server just with a `print` function.

4.3.2.4. *adder.cgi.* It is a small further step to processing dynamic input. Try filling out and submitting the adder form one more time, `http://localhost:8080/adder.html`. This time notice the URL at the top of the browser page *when the result is displayed*. You should see something like the following (only the numbers should be the ones *you* entered):

`http://localhost:8080/adder.cgi?x=24&y=56`

This shows one mechanism to deliver data from a web form to the CGI script that processes it. The names x and y are used in the form (as we will see later) and the data you entered is associated with those names. In fact a form is not needed at all to create such an association: If you directly go to the URLs `http://localhost:8080/adder.cgi?x=24&y=56` or `http://localhost:8080/adder.cgi?x=-12345678924&y=33333333333`, you get arithmetic displayed without the form. This is just a new input mechanism into the CGI script.

You have already seen a program to produce this adder page from inside a regular Python program taking input from the keyboard. The new CGI version, adder.cgi, only needs to make a few modifications to accept input this way from the browser. New features are commented in the source and discussed below. The new parts are the `import` statement through the `main` function, and the code after the end of the `fileToStr` function. *Read* at least these new parts in the source code shown below:

```
#!/usr/bin/python

import cgi   # NEW

def main(): # NEW except for the call to processInput
    form = cgi.FieldStorage()      # standard cgi script lines to here!

    # use format of next two lines with YOUR names and default data
    numStr1 = form.getfirst("x", "0") # get the form value associated with form
                                      # name 'x'.  Use default "0" if there is none.
    numStr2 = form.getfirst("y", "0") # similarly for name 'y'
    contents = processInput(numStr1, numStr2)   # process input into a page
    print(contents)

def processInput(numStr1, numStr2):  # from additionWeb.py
    '''Process input parameters and return the final page as a string.'''
    num1 = int(numStr1) # transform input to output data
    num2 = int(numStr2)
    total = num1+num2
    return fileToStr('additionTemplate.html').format(**locals())

# the remaining code should become standard in your cgi scripts
def fileToStr(fileName):
    """Return a string containing the contents of the named file."""
    fin = open(fileName);
    contents = fin.read();
    fin.close()
    return contents

try:   # NEW
    print("Content-type: text/html\n\n")   # say generating html
    main()
except:
    cgi.print_exception()                 # catch and print errors
```

First the overall structure of the code:

- To handle the CGI input we import the `cgi` module.
- The main body of the code is in a `main` method, following good programming practice.
- After the definition of `main` come supporting functions, each one copied from the earlier local web page version, additionWeb.py.

  At the end is the new, but standard, cgi wrapper code for `main()`. This is code that you can always just copy. I chose to put the initial `print` function here, that tells the server html is being produced. That mean the `main` method only needs to construct and print the *actual* html code. Also keep the final `try-except` block that catches any execution errors in the program and generates possibly helpful trace information that you can see from your browser. (Writing such error catching code in general is not covered in this introductory tutorial.)

The `main` function has three sections, as in the local web page version: read input (this time from the form), process it, and generate the html output.

- Reading input: The first line of main is a standard one (to copy) that sets up an object called `form` that holds the CGI form data accompanying the web request sent by the browser. You access the form data with statements like the next two that have the pattern:

  `variable` `= form.getfirst(`*`nameAttrib, default`*`)`

  If there is a form field with name *`nameAttrib`*, its value from the browser data is assigned to *variable*. If no value is given in the browser's data for *`nameAttrib`*, *`variable`* is set equal to *default* instead. In this way data associated with names given by the browser can transferred to your Python CGI program. In this program the values associated with the browser-supplied names, 'x' and 'y', are extracted. I use Python variable names that remind you that all values from the browser forms are strings.
- The `processInput` function that is passed the input parameters from whatever source, is exactly the same as in `additionWeb.py`, so we already know it works!
- Output the page. In a CGI script this is easier than with the local web pages: just print it – no need to save and separately display a file! The server captures the "printed" output.

This program can now serve as a template for your own CGI scripts: The only things you need to change are the lines in `main()` that get the input from a web form, and the contents of `processInput`, and the `processInput` part can be written and tested earlier with a local web page. While this is the only Python code, you still need to create an output web page template, and refer to it in the parameter of `fileToStr`.

4.3.3. Errors in CGI Scripts. Before you start running your own CGI scripts on the local server, it is important to understand how different kinds of errors that you might make will be handled.

Syntax errors: You are encouraged to check for syntax errors *inside* Idle, by either going to the Run menu and selecting Check Module, or by using the shortcut Alt-X. If you fail to do this and try running a script with a syntax error, the error trace appears in the *console* window that appears when you start the local server. If you want an illustration, you might try changing adder.cgi, making an error like `impor cgi`, and try using adder.html with the flawed script. (Then fix it and try again.)

Execution Errors: The error trace for execution errors is displayed in your *web browser*, thanks to the final standard code with the `try-catch` block at the end of the CGI script. If you omit that final standard code, you completely lose descriptive feedback: Be sure to include that standard code! You can also illustrate here. Get an indexing error by introducing the statement bad = 'abc'[5] in the main function. (Then take it out.)

Logical Errors: Since your output appears in the web browser, when you produced something legal but other than what you intended, you see in the browser . If it is a formatting error, fix your output page template. If you get wrong answers, check your `processInput`.

We have not covered web forms yet, but rather than bite off too much at once, this is a good time to write your own first CGI script.

Exercise 4.3.3.1. ** Modify Exercise 4.2.1.1 and save it as a CGI script `quotient.cgi,` in the *same* directory where you have `localCGIServer.py` and your output page template. Make quotient.cgi take its input from a browser, rather than the keyboard. This means merging all the standard CGI code from

adder.cgi and the processInput code from your quotientWeb.py. You can keep the same browser data names, x and y, as in adder.cgi, so the main method should not need changes from adder.cgi. Remember to test for syntax errors inside Idle, and to have the local web server running when you run the CGI script in your browser. Since we have not yet covered web forms, test your CGI script by entering test data into the URL in the browser, like by going to links `http://localhost:8080/quotient.cgi?x=24&y=56` and `http://localhost:8080/quotient.cgi?x=36&y=15`. After trying these links, you can edit the numbers in the URL in the browser to see different results.

4.3.4. Editing HTML Forms. This section is a continuation of Section 4.1.2. It is about HTML editing, not Python. HTML forms will allow user-friendly data entry for Python CGI scripts. This is the last elaboration to allow basic web interaction: Enter data in a form, submit it, and get a processed result back from the server.

The initial example, adder.html, used only two text fields. To see more common form fields, open `http://localhost:8080/commonFormFields.html`. (Make sure your local server is still running!)

To allow easy concentration on the data sent by the browser, this form connects to a simple CGI script `dumpcgi.cgi`, that just dumps all the form data to a web page. Press the submit button in the form, and see the result. Back up from the output to the previous page, the form, and change some of the data in all kinds of fields. Submit again and see the results. Play with this until you get the idea clearly that the form is passing on your data.

To play with it at a deeper level, open this same file, the www example commonFormFields.html, in *Kompozer*. The static text in this page is set up as a tutorial on forms in Kompozer. Read the content of the page describing how to edit the overall form and each type of individual field. Textbooks such as the Analytical Engine give another discussion of some of the attributes associated with each field type. Read the static text about how to edit individual fields, and change some field parameters, save the file and reload it in your browser, and submit again. If you change the name or value attributes, they are immediately indicated in the dumped output. If you change things like the text field size, it makes a change in the way the form looks and behaves. You can return to the original version: An extra copy is saved in commonFormFieldsOrig.html.

Now open adder.html in Kompozer. Switch to the Source view. This is a short enough page that you should not get lost in the source code. The raw text illustrates another feature of html: attributes. The tag to start the form contains not only the tag code *form*, but also several expressions that look like Python assignment statements with string values. The names on the left-hand side of the equal signs identify a type of *attribute*, and the string value after the equal sign gives the corresponding *value* for the attribute. The tag for many kinds of input fields is `input`. Notice that each field includes `name` and `value` attributes. See that the 'x' and 'y' that are passed in the URL by the browser come from the names given in the HTML code for the corresponding fields.

Kompozer and other web editors translate your menu selections into the raw html code with proper attribute types. This high level editor behavior is convenient to avoid having to learn and debug the exact right html syntax! On the other hand, using pop-up field editing windows has the disadvantage that you can only see the attributes of one field at a time. Particularly if you want to modify a number of name or value attributes, it is annoying that you need a number of mouse clicks to go from one field to the next. If you only want to modify the *values* of existing attributes like `name` and `value`, it may be easier to do in the source window, where you can see everything at once. Making syntax errors in not very likely if you *only* change data in quoted value strings.

The action URL is a property of the entire form. To edit it in Kompozer, right click inside the form, but *not* on any field element, and select the bottom pop-up choice, Form Properties. Then you see a window listing the Action URL and you can change the value to the name of the CGI script that you want to receive the form data. When you create your own web form, I suggest you make the initial action URL be dumpcgi.cgi. This will allow you to debug your form separate from your CGI script. When you have tested that your web form has all the right names and initial values, you can change the action URL to your CGI script name (like quotient.cgi), and go on to test the combination of the form and the CGI script!

Exercise 4.3.4.1. ** Complete the web presentation for quotient.cgi of Exercise 4.3.3.1 by creating a web form quotient.html that is intelligible to a user and which supplies the necessary data to quotient.cgi.

Be sure to test the new form on your local server! Remember that you must have the local server running first. You must have all the associated files in the same directory as the server program you are running, and you cannot just click on quiotient.html in a file browser. You must start it from the the URL `http://localhost:8080/quotient.html`, that specifically refers to the server localhost.

Exercise 4.3.4.2. ** Make a simple complete dynamic web presentation with a CGI script that uses at least *three* user inputs from a form. The simplest would be to just add three numbers instead of two. Call your form dynamic.html. Call your CGI script dynamic.cgi. Call an output template dynamicTemplate.html. remember the details listed in the previous exercise to make the results work on localhost. After the server is started and you have all the files, go to `http://localhost:8080/dynamic.html`.

The Summary Section 4.4 starts with the overall process for creating dynamic web pages.

4.3.5. More Advanced Examples. One of the advantages of having a program running on a public server is that data may be stored centrally and augmented and shared by all. In high performance sites data is typically stored in a sophisticated database, beyond the scope of this tutorial. For a less robust but simpler way to store data persistently, we can use simple text files on the server.

The www example page namelist.html uses namelist.cgi to maintain a file namelist.txt of data submitted by users of the page. You can test the program with your local Python server. It is less impressive when you are the only one who can make changes! You may also try the copy on the public Loyola server, `http://cs.luc.edu/anh/python/hands-on/examples/www/namelist.html`. The local source code is documented for those who would like to have a look.

You also may want to look at the source code of the utility script you have been using, dumpcgi.cgi. It uses a method of getting values from the CGI data that has not been discussed:

```
val = form.getlist(name)
```

This method returns a list of values associated with a name from the web form. The list many have, 0, 1, or many elements. It is needed if you have a number of check boxes with the same name. (Maybe you want a list of all the toppings someone selects for a pizza.)

Both dumpcgi.cgi and namelist.html add an extra layer of robustness in reflecting back arbitrary text from a user. The user's text may include symbols used specially in html like '<'. The function `safePlainText` replaces reserved symbols with appropriate alternatives.

The examples in earlier sections were designed to illustrate the flow of data from input form to output page, but neither the html or the data transformations have been very complicated. A more elaborate situation is ordering pizza online, and recording the orders for the restaurant owner. You can try `http://localhost:8080/pizza1.cgi` several times and look at the supporting example www files pizza1.cgi, pizzaOrderTemplate1.html, and the simple pizzaReportTemplate.html. To see the report, the owner needs to know the special name `owner777`. After ordering several pizzas, enter that name and press the Submit button again.

This script gets used in two ways by a regular user: initially, when there is no order, and later to confirm an order that has been submitted. The two situations use different logic, and the script must distinguish what is the current use. A hidden variable is used to distinguish the two cases: when pizza1.cgi is called directly (not from a form), there is no pastState field. On the other hand the pizzaOrderTemplate1.html includes a hidden field named pastState, which is set to the value 'order'. (You can confirm this by examining the end of the page in Kompozer's source mode.) The script checks the value of the field pastState, and varies its behavior based on whether the value is 'order' or not.

The form in pizzaOrderTemplate1.html has radio buttons and check boxes hard coded into it for the options, and copies of the data are in pizza1.cgi. Keeping multiple active copies of data is not a good idea: They can get out of sync. If you look at the source code for pizzaOrderTemplate1.html, you see that all the entries for the radio button and check box lines are in a similar form. In the better version with altered files pizza.cgi and pizzaOrderTemplate.html (that appears the same to the user), the basic data for the pizza options is only in one place in pizza.cgi, and the proper number of lines of radio buttons and check boxes with the right data are generated dynamically.

A further possible elaboration would be to also allow the restaurant manager to edit the size, cost and topping data online, and store the data in a file rather than having the data coded in pizza.cgi, so if the

manager runs out of a topping, she can remove it from the order form. this change would be a fairly elaborate project compared to the earlier exercises!

Final www examples are a pair of programs in real use in my courses. To illustrate, you can try the sample survey, `http://localhost:8080/pythonTutorialsurvey.html`. Run it several times with different responses. Forms can be set up like this one to link to the www example CGI script surveyFeedback.cgi, which will save any number of responses to the survey. After getting responses you can start the Idle shortcut in the www example directory and run the regular Python program, readFeedback.py, which is also in the www example directory: At the prompt for a survey base name, enter *exactly*:

```
pythonTutorial
```

Then the program prints out all the survey feedback, grouped in two different ways. It documents the use of a couple of modules not introduced in this tutorial, but the rest just uses ideas from the tutorial, including considerable emphasis on dictionaries and string processing.

4.4. Summary

(1) The Overall Process for Creating Dynamic Web Pages
Making dynamic web pages has a number of steps. I have suggested several ways of decoupling the parts, so you can alter the order, but if you are starting from nothing, you might follow the following sequence:
   (a) Determine the inputs you want to work with and make a web form that makes it easy and obvious for the user to provide the data. You may initially want to have the form's action URL be dumpcgi.cgi, so you can debug the form separately. Test with the local server. When everything seems OK, make sure to change the action URL to be the name of the CGI script you are writing. [4.3.4]
   (b) It is easier to debug a regular Python program totally inside Idle than to mix the Idle editor and server execution. Particularly if the generation of output data is going to be complicated or there are lots of places you are planning to insert data into an output template, I suggest you write the `processInput` function with its output template first and test it without a server, as we did with `additionWeb.py`, providing either canned input in the main program, or taking input data from the keyboard with input, and saving the output page to a local file that you examine in your webbrowser. [4.2.1]
   (c) When you are confident about your `processInput` function, put it in a program with the proper cgi skeleton, and add the necessary lines at the beginning of the `main` function to take all the CGI script input from the browser data. [4.3.2.4]
   (d) Finally test the whole thing with the local server.
   (e) If you have an account on a public server, it should not take much more work than just uploading your files to make your creation available to the whole world. You may have a public server with a different configuration than the Loyola server. If so see this note:[1]

(2) Markup: Plain text may be *marked up* to include formatting. The formatting may be only easily interpreted by a computer, or it may be more human readable. One form of human-readable markup is hypertext markup language (HTML). [4.1.1]
   (a) HTML markup involves tags enclosed in angle braces. Ending tags start with '/'. For instance `<title>Computer Science</title>`.
      (i) Tags may be modified with attributes specified similar to Python string assignments, for example the text input field tag,
      `<input value="red" name="color" type="radio">`
   (b) Modern editors allow HTML to be edited much like in a word processor. Two views of the data are useful: the formatted view and the source view, showing the raw HTML markup.

[1]The tutorial assumed a server configured as follows: html pages and CGI scripts can all be in the same directory, and the CGI scripts end with .cgi. This is the convention on Loyola's Computer Science public student server. Another common configuration is that scripts all go in a cgi-bin directory, where they just have the normal .py suffix. If you have a server with the latter configuration, your action URLs will be of the form cgi-bin/*someScript*.py. Depending on the server configuration the current directory may or may not be cgi-bin while the script executes. That may mean you need a path before the file names for your output templates, or your need to be careful what directory referenced files end up in. If you are making arrangements for your own site on a public server, be sure to check with your system administartor to find out what the conventions are.

(3) Python and HTML: Since HTML is just a text string, it can easily be manipulated in Python, and read and written to text files. [4.2.1]
(4) The `webbrowser` module has a function `open`, that will open a file or web URL in the default browser: [4.2.1]
`webbrowser.open(`*filename*`)`
(5) Common Gateway Interface (CGI). The sequence of events for generating a dynamic web page via CGI: [4.3.1]
 (a) The data a user types is handled directly by the browser. It recognizes forms.
 (b) The user presses a Submit button. An action is stored in the form saying what to do when the button is pressed.
 (c) In the cases we consider in this tutorial, the action is given as a web resource, giving the location of a CGI script on some server. The browser sends the data that you entered to that web location.
 (d) The server recognizes the page as an executable script, sees that it is a Python program, and executes it, using the data sent along from the browser form as input.
 (e) The script runs, manipulates the input data into some results, and puts those results into the text of a web page that is the output of the program.
 (f) The server captures this output from the program and send it back to the user's browser as a new page to display.
 (g) The results appear in the user's browser.
(6) The `cgi` Module
 (a) Create the object to process CGI input with [4.3.2.4]
 `form = cgi.FieldStorage()`
 (b) Extract the first value specified by the browser with name *nameAttrib*, or use *default* if no such value exists [4.3.2.4]
 variable `= form.getfirst(`*nameAttrib*`,` *default*`)`
 (c) Extract the `list` of all values specified by the browser associated with name*, nameAttrib* [4.3.5]
 listVariable `= form.getlist(`*nameAttrib*`)`
 This case occurs if you have a number of checkboxes, all with the same name, but different values.
(7) Local Python Servers.
 (a) Python has modules for creating local testing servers that can handle static web pages and Python CGI scripts.[4.3.1]
 (b) Different kinds of errors with CGI scripts are handled different ways by a local Python server. [4.3.3]
(8) A comparison of the various types of files used in web programming, listing the different ways to edit and use the files, is given in Section 4.1.3.